Silesian University of Technology

Faculty of Automatic Control, Electronics and Computer Science

Final Project

GPU-accelerated stochastic simulator engine for PRISM model checker.

Author: Marcin Copik
Supervisor: prof. Tadeusz Czachórski
Consultant: dr inż. Artur Rataj

Gliwice, February 2014

Acknowledgements

I would like to thank Prof. Tadeusz Czachórski for his support and supervision of this dissertation.

My deepest thanks go to Dr. Artur Rataj, who has introduced me into the field of model checking, presented the PRISM tool and proposed to work on this project.

I am very grateful for the possibility to work on PRISM, provided to me by Dr. Dave Parker from the University of Birmingham. His kind assistance, offered before and during this project, is invaluable.

Finally, I would like to thank Dr. Dominik Spinczyk, my first supervisor at the University, for the experience and knowledge which I was able to gain during our collaboration and for encouraging me to develop my skills.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Verification by model checking 3

1.3 PRISM model checker 4

1.4 Aims of the project 5

1.5 Organization of this dissertation 5

2 Theoretical background **7**

2.1 Model checking 7

2.1.1 Formal methods 7

2.1.2 Model checking 9

2.1.3 Probabilistic model checking 18

2.1.3.1 Models for probabilistic model checking 18

2.1.3.2 Logics for probabilistic model checking 21

2.1.4 Statistical probabilistic model checking 23

2.2 General-purpose computing on GPU 28

2.2.1 OpenCL 28

2.2.1.1 OpenCL memory model 30

2.2.1.2 OpenCL C programming language 32

2.2.1.3 OpenCL on GPU 33

3 Problem Analysis **35**

3.1 Model Specification in PRISM 35

3.1.1 Discrete-time Markov Chain 36

3.1.2 Continuous-time Markov Chain 38

3.2 Approximate Probabilistic Model Checking 40

3.2.1 Suitable Properties for Simulation 41

3.2.2 Property Evaluation 42

3.3 Current Simulator Engine in PRISM 44

3.4 Requirements for New Simulator Engine . . . 45
3.4.1 Structural Requirements . . . 45
3.4.2 Verification Requirements . . . 46
3.4.3 User Interface Requirements . . . 46
3.5 Related work . . . 47
3.5.1 The Approximate Probabilistic Model Checker . . . 47
3.5.2 Ymer . . . 47
3.5.3 VeStA . . . 47
3.5.4 PRISM-U2B . . . 48

4 Design and Implementation **49**
4.1 Prism Simulator Engine . . . 49
4.2 GPU Simulator Engine . . . 52
4.3 Integration with User Interfaces . . . 53
4.3.1 Command-Line Interface . . . 53
4.3.2 Graphical User Interface . . . 54
4.4 KernelGenerator Library . . . 54

5 OpenCL Implementation **59**
5.1 Kernel design . . . 59
5.1.1 State Vector Representation . . . 60
5.1.2 Transition Representation . . . 61
5.1.3 Synchronized Update . . . 62
5.1.4 Loop Detection . . . 62
5.2 Pseudo-random Number Generators . . . 64
5.2.1 mwc64x . . . 64
5.2.2 OpenCL PRNG Library . . . 64
5.2.3 Random123 . . . 64

6 Testing **67**
6.1 Testing Setup . . . 67
6.2 Case Studies . . . 68
6.2.1 Probabilistic Contract Signing Protocol . . . 68
6.2.2 NAND Multiplexing . . . 70
6.2.3 Tandem Queueing Network . . . 70
6.3 Analysis . . . 72
6.3.1 Correctness of results . . . 72

6.3.2 Performance differences 72

7 Conclusions **75**

7.1 Summary of Project 75

7.2 Deficiencies and Future Work 76

A Case Studies **77**

A.1 Probabilistic Contract Signing Protocol 77

A.2 NAND Multiplexing 79

A.3 Tandem Queueing Network 81

Bibliography **82**

List of Figures

1.1 Markov chain for communication protocol. 3

2.1 Model checking schema. 9
2.2 Basic CTL operators. 15
2.3 Overview of statistical model checking. 23
2.4 Schematic view of an OpenCL device. 30
2.5 Schematic view of memory spaces in OpenCL. 31

3.1 DTMC built from model in Listing 3.1. 37
3.2 CTMC built from model in Listing 3.2. 39
3.3 Simulation panel in PRISM GUI. 45

4.1 Data flow between PRISM and SimulatorEngine. 50
4.2 Sampler class hierarchy. 51
4.3 SimulationMethod class hierarchy. 52
4.4 Data flow with ModelCheckInterface. 52
4.5 Runtime and device interfaces in GPU Simulator Engine. 53
4.6 Modified simulation panel in PRISM GUI. 55
4.7 Basic structure of memory classes in the KernelGenerator library. 56
4.8 Basic structure of expression classes in the KernelGenerator library. 57

Chapter 1

Introduction

1.1 Motivation

Modern society has put new challenges for computer systems, which have transformed from strictly scientific, business or military tool to a basis of almost each technological product. Nowadays software systems play crucial role in applications where safety and reliability are the most important requirements. Automotive industry heavily uses embedded systems for controlling airbags, braking, electronic traction control systems, etc. Non-invasive diagnostic methods in medicine demand sophisticated electronic devices, which are obliged to provide accurate results with great care for patients health.

The history of computer systems remembers not only an uncounted number of pitfalls, but there has been a few cases where developer's mistake had horrible consequences.
The first launch of Ariane 5, a rocket created by European Space Agency for delivering payloads on Earth's orbits, was a complete failure. The rocket self-destruction after 37 seconds was a result of serious software malfunction. The main reason of this catastrophe was an unhandled exception caused by conversion of 64-bit floating point value to 16-bit signed integer. The loss was estimated to over $370 million.
The London Ambulance Service Computerized Dispatch System is considered as a good example of a software failure. The aim of the project was to fully automatize the procedure of dispatching an ambulance to the place of emergency call. It was supposed to deal with over 2000 calls per day and fleet of 750 vehicles. The effects of system's malfunctions were severe: delays with accepting calls, late arrival of ambulances, duplicate or none ambulance showing up at incident place. Two deaths have been proved as a direct consequence of failure[1].

Software engineering has developed many crucial methods for creating reliable products. Several testing techniques are used widely in software companies and the unit testing has become a standard which is required from every newly hired developer. Software testing, peer review and other procedures have to meet the requirements imposed by customers - solid software created as fast as

it is possible. It would be unrealistic to expect from these methods to completely erase bugs and detect malfunctions in a system. Popular in the extreme programming methodology maxim: "test everything that can break" shows the main source of undetected bugs: there is no way to test each execution path, so one must look for these methods or modules which are the most likely to create problems and crash a system. It is quite similar to famous Dijkstra's observation "testing shows the presence, not the absence of bugs" [2].

The economic motivation for formalizing software development process is strong: the cost of fixing an error discovered after installation is about 500 times more than the cost of removinga bug which was found during early design phase [3, page 275]. National Institute of Standards and Technology in 2002 estimated the total cost of software bugs and failures in the USA at $59,5 billion annually, which is 0.6 percent of the US GDP [4]. More recent study conducted at Cambridge Judge Business School estimated the worldwide cost of time spent by developers on finding and fixing bugs and cost of delay in product delivery to $312 billion annually [5]. However, the study was not published, only the results were announced as news information, and the research was made for Cambridge start-up Undo which produces technology called 'reversible debugger', so it is hard to evaluate the correctness of this number.

In addition to natural limitations of testing techniques, there coexist the problem of system's intricacy. Technological progress and increasing importance of computer applications in business or industry creates demand for bigger, more advanced and customized software solutions and their market lifetime was decreasing through the last decade. It is not too temerarious to say that current complexity of software of computer systems makes it one of the most complicated human creations.[1] Currently popular trend in software engineering is to project systems as a connection of independent modules and focus on interfaces which describe communication and module responsibility, as the most crucial part in the whole design. This practice simplifies the architecture and reduces probability of problems and bugs created by unnecessary connections which are likely to be broken after making changes in the code; it will also reduce time and cost because modules may be developed and tested separately. However, the working modules are going to change their internal state independently, which will influence the results and potential problems in communication and cooperation of modules; that heavily complicates predicting systems behaviour. Further inventions which include: parallel programming for multi-core processors, concurrency with locks, synchronization and race conditions, distributed computing and massive communications over different types of networks, generate new error types and critical operations, which may lead to severe system failure. It is almost impossible to test each situation and find all bugs in systems which behaviour is nondeterministic.

[1] Brooks' important conclusion: "One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate of man's handiworks."[6]

1.2 Verification by model checking

The process of designing complex, reliable and safe systems requires an enhancement of the standard engineering tools. This goal was reached by introducing the *formal methods*, discussed in chapter 2.1.1. One of the most notable formal methods is model checking, proposed as a technique for automatic verification of concurrent systems. In the process of model checking, a system is expressed as *a finite-state automata*, containing a set of *states* and a collection of *transitions*. State captures the system behaviour and transition describes how the system reacts to events, in terms of changing its internal state. By exploring the whole model, it is possible to verify if certain requirements hold - *model checker* is responsible for effective and correct conducting of this operation. Chapter 2.1.2 describes in detail the process of modeling, specifying requirements and checking their correctness.

To make this introduction clear and understandable, we decided to present new concepts on the example of a simple communication protocol[2]. This protocol operates over an error-prone channel, i.e. in which message may be lost. The process of sending a message is characterized by few rules:

1. After one unit of time, system **tries** to send a message.
2. With probability 0.98, the message is **successfully** delivered and system awaits for the next task.
3. With probability 0.01, sending the message **fails**, after one step the sending is repeated.
4. With probability 0.01, channel is busy and system awaits for one unit of time.

Figure 1.1 depicts an automaton constructed for this protocol.

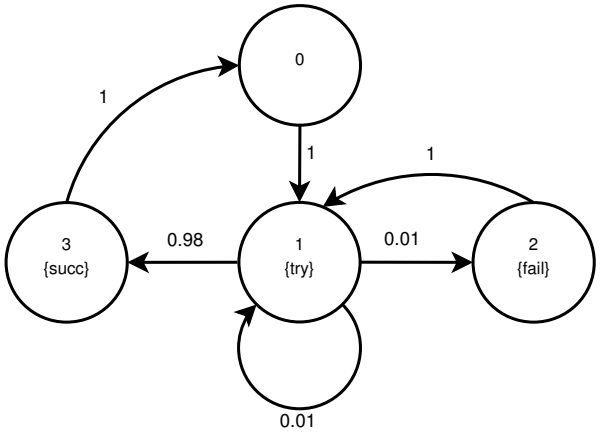


Figure 1.1: Markov chain for communication protocol.

Formally, the automaton where 'outgoing' transitions from state are given by probabilities and the time is interpreted as a sequence of natural numbers, is known as the *discrete-time Markov chain*

[2]Example presented in [3],[7].

(DTMC), presented in chapter 2.1.3.1.

This protocol is an example of *stochastic system* i.e. system where decisions are characterized with the probability distribution. *Probabilistic model checking* is an extension of a traditional approach, where it is allowed to compute the probability that a certain property holds in a model. To express the requirements, we will use *temporal logics* and for a stochastic model like DTMC, it is convenient to use probabilistic temporal logic *Probabilistic Computational Tree Logic* (PCTL), defined in chapter 2.1.3.2.
For example, if we want to compute probability that a message is eventually delivered, we will use temporal operator **F**, interpreted as 'event will occur in the **F**uture':

$$\boldsymbol{P}_{=?}\,[\boldsymbol{F}\ succ]$$

The message will not be delivered only if the system will be visiting states *try* and *fail* the infinitely number of times. Obviously, the probability of that event is equal to zero.
We may include timing constraints into requirements, e.g. 'the probability of a message being delivered withing 5 steps' is expressed as:

$$\boldsymbol{P}_{=?}\,[\boldsymbol{F} \leq 5\ succ]$$

The probabilistic temporal logics PCTL and *Continuous Stochastic Logic* (CSL) are extensions of Computational Tree Logic (CTL), defined in 2.1.2.

Unfortunately, models in real-world applications of model checking are rarely so simple. Usually they contain millions of states and it is not possible to process them directly, due to insufficient size of memory. To avoid this problem, researchers proposed many different ideas and one of them is an application of the Monte Carlo method. A simulator generates many random paths in the model and evaluation of property in samples allows estimation of real probability. This approach is known as the *statistical probabilistic model checking* and it is presented in chapter 2.1.4.

1.3 PRISM model checker

PRISM[8] is a probabilistic model checker, developed at the University of Birmingham. PRISM allows the analysis of stochastic and nondeterministic systems, defined with the PRISM language and modelled as Markov chains (DTMC or continuous-time Markov Chain), Markov decision process (MDP), probabilistic automata(PA) or probabilistic timed automata (PTA). The property specification language contains a wide range of temporal logics, including Lineral Temporal Logic (LTL), mentioned earlier CTL, CSL, PCTL, and others.
In addition to analytical algorithms, implemented with symbolic data structures, PRISM incorporates simulation engine, used as a debugging tool for models or for performing the statistical model

checking.
PRISM is released under the GNU General Public License[9].

1.4 Aims of the project

Existing simulator, described with more details in Chapter 3, provides intuitive graphical user interface for debugging models. It is considered as a valuable tool, but its internal implementation in Java is strictly sequential. Random sampling is a fully parallel task, therefore, the effectiveness of statistical model checking may be greatly improved. The main goal of this project is to implement new, parallel simulator engine for PRISM that utilizes the power of OpenCL to perform random sampling on many devices, especially on the Graphical Processors Units (GPUs). The main objectives are:

- Desing and implement OpenCL kernel containing:
  - algorithms and data structures for path generation.
  - methods for property verification.
- Refactor existing code, create common interfaces to work equally with both simulator engines.
- Design efficient connection with implemented in PRISM methods for statistical model checking.
- Extend graphical user interface, providing selection of OpenCL device and configuration of simulator.

1.5 Organization of this dissertation

Next chapter provides theoretical background for the project, including description of model checking and brief introduction into computing on GPGPU. Chapter 3 describes PRISM modelling language and the existing simulator, which is followed by the list of requirements for new simulator and a review of related solutions.
The design of simulator and class hierarchy is discussed in Chapter 4. Chapter 5 presents the most important and curious parts of OpenCL kernels.
Chapter 6 contains verification results of selected case studies, including times of model checking.
Chapter 7, consisting of a brief summary and list of possible future improvements, is followed by Appendix A with full PRISM models from case studies used in Chapter 6.

Chapter 2

Theoretical background

Chapter Abstract

This chapter contains theoretical background for developing the simulator engine. It is split into two parts: the general, full introduction into model checking and the sketch of parallel programming on General Purpose Graphic Processor Units.
Chapter 2.1 begins with an introduction to formal methods, section 2.1.3 presents general view of model checking and section 2.1.4 introduces probabilistic model checking, an extended approach for stochastic systems, with full description of model types and logics used in this project. Finally, statistical approach to model checking is discussed.

2.1 Model checking

2.1.1 Formal methods

Intuitive definition of formal methods is that they are strictly mathematical techniques for specification and verification of computer systems, both software and hardware. The application of mathematical logic and various different methods, e.g. theory of automata, allows proving correctness, security and reliability of a system. Formal methods are rigorous tool, and it is impractical and often impossible to formally specify each detail. A curious task is to find these part of system which require formalization. An example ofa module, which should not be verified using formal methods, is the graphical user interface.
The application of formal methods requires three parts:

- **system specification** - typically a set of formulas or a model.
- **formalized requirements** - with dedicated description language.
- **verification method** - checking whether the system satisfies requirements.

"Logic in Computer Science" [10] provides a simple set of criterias for classification of verification methods:

Proof-based vs model-based

Main difference between these two methods lies in system description - a proof-based method tries to derive a proof that specification formula is satisfied by the system, which is set of logic formulas. It may require guidance from human operator.

Model-based approach checks if system represented by some kind of model satisfies specification formula. It could be fully automatized for finite models.

Degree of automation

Formal methods may vary in that aspect, from completely manual to completely automatic.

Full- vs. property-verification

Specification may describe whole system's behaviour, but it is rather uncommon.

Intended domain of application

Formal methods are applicable for many different types of systems and that includes hardware, operating systems, concurrent programs etc.

Pre- vs. post-development

Systems may be verified during early design, later stages of development or before installation. However, it is beneficial to specify and test requirements before implementation.

For example, *automated theorem proving* is a popular approach to verification, where system and requirements are specified with logical sentences. The computer, usually guided by human, tries to deduce from a specification a proof that properties are valid. In terms of above classification, it is a proof-based, partly automated formal method working on full system specification.

The first well-known case study is the IBM's CICS/ESA system[11]. Z notation, formal specification language proposed by Jean-Raymond Abrial in 1977, was introduced into development of new release of CISC/ESA, which included new and modified code (268,000 lines) and restructuring for future versions. Specification of about 15% of new source code improved detection of problems during first design phase and reduced number of errors in further development phases, mainly during functional verification and system tests. Programmers had to spend shorter time on fixing problems and that have resulted in drop of development costs by 9%. Another important conclusion from this study is that introduction of formal methods have positively affected general system quality.

Formal methods had a huge impact on the development of safe and reliable computer systems. The advance in algorithms and computational possibilities made them a useful and practical technique, widely used for erasement of designers' mistakes. To emphasize this influence, a noteworthy conclusion from report prepared by NASA contractor[12, chap.3][1] is presented:

[1] Report is based on FAA results (the US Federal Aviation Administration)

Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers.

2.1.2 Model checking

Model checking has developed as verification technique for concurrent[13] and reactive systems[14] i.e. systems, which work continuously and react for changes in their environment (it is not possible to reason about their behaviour using only initial input and output). In terms of classification from previous section, model checking could be described as a method that is: fully automated, property-verification, based on finite state model and used both for software and hardware verification.
Model checking has made an impressive progress over three decades of development, beginning in the early 80's with pioneering work by E. M. Clarke with E. A. Emerson (application of temporal logic and finite models to construction of concurrent programs)[15] and J. P. Queille with J. Sifakis (designing concurrent systems, verified with specification written in formulas of time logic)[16].
The importance of model checking for computer science was emphasized by rewarding Clarke, Emerson and Sifakis with Turing Award in 2007, for "their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries."[17]

The main idea behind model checking is: given model and property, perform exhaustive, brute-force exploration of the model to find all states that satisfy constraint specified in property. The schematic view in Figure 2.1 shows that, if exploring all possible paths through model indicates that requirement does not hold, model checker will generate a counterexample, which will make much easier finding the source of problem; it is an important advantage over other formal methods.

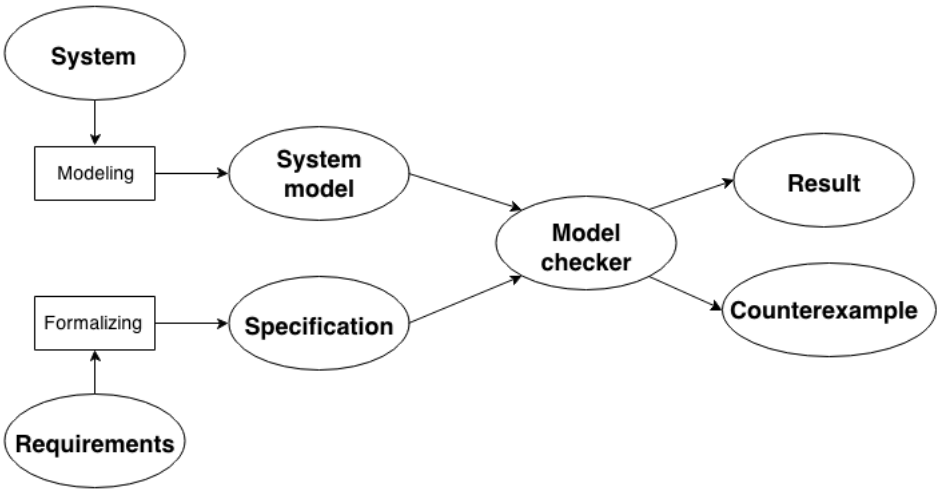


Figure 2.1: Model checking schema.

The modelling phase represents 'system specification' component from formal methods, 'formalization' corresponds obviously to 'formalized requirements' and the model checker speaks for 'verification method'.

System specification

Appearance of word "model" in technique name underlines how it is crucial in the whole process of analysis, how important for quality and correctness of results is quality of the model. We have categorized model checking as formal method aimed at property verification, because the examination of one selected attribute does not require the user to prepare full and detailed specification of each module in system.

The system description is based on the concept of *state*, which is defined as a collection containing the values of all variables that are necessary to define system. State is essential not only for storing information about system's status at a concrete time, but it is used also to determine how the system will react to changes and actions. For this purpose the concept of *transition* is introduced; transition describes system's reaction for some particular change and because of that two states are required: one to give information about system before action and one to express which system parameters have changed. Formally it is defined as a pair of states. Model checking is concentrated on exploring system's behaviour through time and that is why the intuitive concept of system's execution must be defined. The *path*[2] through the model stands for an infinite sequence of states, with objection to existence of transition between each two neighbouring states.
The graphical interpretation is a directed graph, in which the nodes represent system states, and transitions are depicted with directed edges, oriented from 'input' to 'output' state.
There coexist many similar types of state-transition system; for this introduction we have chosen widely-used Kripke structure[14]:

Definition Let AP be set of atomic propositions. A *Kripke structure* M over AP is a four tuple $M = (S, S_0, R, L)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be left-total, that is
$$\forall_{s \in S} \exists_{s' \in S} R(s, s').$$
4. $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions, which are true in that state.

[2]There exist commutative terms in the literature, e.g. *computation* in [14]

Definition A *path* in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ such that $s_0 = s$ and $\forall_{i \geq 0} R(s_i, s_{i+1})$. i-th state in path π is defined as $\pi(i)$.

The requirement for relation R to be left-total excludes dead end states, i.e., states that do not have any outgoing transitions; unreachable and dead end states are not necessary to properly understand process of model checking. However, for formal reasons, the definition of Kripke structure may not make any assumptions about totalness of R[18].

Nonetheless, common practice is to create an automata, on which model checking methods will be applied, from more high-level system descriptions[3]. Examples of that kind modelling tools are:

- **Petri nets**

  Petri net is a mathematical modelling tool, commonly used to describe distributed systems. An example of that kind of model checker is ALPiNA[19], which verifies models based on the Algebraic Petri Net.

- **process algebras**

  Queille and Sifakis in their originate paper[16] specified system using high level language inspired from CSP(Communicating Sequential Processes), process algebra proposed in 1978 by C.A.R. Hoare.

- **dedicated languages**

  Used in this dissertation PRISM has its own language, where model is described as a parallel composition of modules, each one containing several pairs of update with its guard.

- **traditional programming languages**

  - **SPIN model checker**

    SPIN[20] uses Promela language but enables embedding code in C programming language

  - **Java PathFinder**

    Java PathFinder[21], a model checker developed at NASA, verifies programs written for Java Virtual Machine by analyzing Java bytecode

  - **Java2TADD**

    Java2TADD[22] translates program written in the subset of Java language into a timed automata and generates properties(e.g. variable limits), as an input for model checker

System specification has to be accurate. Avoidance of details, which may seem to be negligible, may result in constructing a model which exploration will not show a minacious path. The erroneous underspecification can lead to 'losing' crucial state.

[3] In reality, modern model checkers use more tricky and efficient data types; more details on this topic are given in the end of this paragraph when *symbolic model checking* is defined.

On the other hand, the problem of modelling huge, complex systems was a relevant challenge from the beginning of model checking. Including more and more independent parameters into system's description causes combinatorial increase of the number of possible states. Hence the model becomes too large to be handled due to insufficient size of computer memory.

This undesirable effect is easily observable in environments like PRISM, where state vector contains all variables declared in model, e.g. Boolean and integer with constant range[4] variables, then the maximal number of states is equal to product of all variable ranges[5]. The state space starts to grow exponentially, with each addition of new variable or module. This problem is known as the **state-space explosion** and it has been keeping down applications of model checking to real world problems through its thirty years of development, though it was a 'driving force'[13] behind many new techniques and algorithms. State-of-the-art model checker can handle state-space much larger than twenty years ago, but still this problem remains one of the most important challenges for model checking. Researchers have proposed many solutions to overcome this problem. We decided to introduce only **symbolic model checking** for better clearance and description, because PRISM is a symbolic model checker[6].

The advance of this technique begins in the 1987, when Kennethe McMillan, graduate student at Carnegie University, comes up with idea to change representation of model from a finite-state automata to *ordered binary decision diagrams*(OBDDs)[7]. Efficient algorithms for manipulating OBDDs have been provided, enabling operations on them in polynomial time, and researchers were able to apply directly original methods on model based this diagrams. The model regularity captured by this representation enabled verification of much more complex systems, protocols and applications, starting from 10^{20} states, which was a huge step forward[14].

McMillan has continued work on symbolic approaches to model checking and one of the results of his PhD thesis[24] was model checker SMV, which have become a widely-used tool for systems verification. There has been many applications of SMV to real-world problems and the most commonly presented in literature is the case of IEEE Futurebus+ Standard(IEEE 896)[8]. First attempt to verify formally its correctness was made in 1992, when the model of processors connected by bus with shared memory was tested for cache consistency. Model containing 10^{30} states, built from 2,300 lines of SMV code, was used to find severe errors which forced revision of standard[25].

The symbolic representation of model has become very successful concept and almost each modern

[4]PRISM also supports clocks, but they are only used in timed automata(TA)/probabilistic timed automata(PTA) and the state space is computed there in different way.

[5]Obviously, PRISM processes model quite differently and excludes all states that are not achievable

[6]Interested reader may look into chapters 1.5-1.7 of Clarke's textbook[14], where he will find presentation of other approaches to state-space explosion, with propositions for further reading.

[7]Originally proposed by R.E.Bryant in 1986[23]. OBDDs are very widely used data structure and Bryant's originate paper is one of the most cited articles in the history of computer science.

[8]Futurebus+, as a successor of Futurebus, was not standardized after showing on market, but was designed by standardizing committees; problematic and long revisions were on of the reasons of its failure - Futurebus+ never replaced the traditional hardware and its usage was very limited(e.g. psychics computation centres)s.

model checker performs symbolic verification, using OBDDs and other data structures instead of traditional state-transition representation.

Subsection 2.2.2 in next section presents and distinctly describes Markov chains, model types used in this project.

Formalized requirements

The model checking s developed two different methods to describe properties: *behavior-based* and *logic-based*[9]. Behavior-based approach relies in equal representation of both the model and properties, which should 'behave' in the same way. Verification for property stated as an automata comes through *bisimulation*, i.e., checking that model automata may simulate every step of property automata[26]; it is rarely used, comparing to logic-based approach.

The majority of applications, algorithms and model checkers are based on logic-based approach. *Temporal logic* has been used to specify properties since the beginning of model checking. This concept differs from traditional propositional or predicate logic by allowing to reason about time, which changes the nature of the value of logical formula - it is no longer *statically* true or false in model, but its evaluation becomes *dynamical*, changes during model execution. Firstly it was proposed by philosophers in 1950s and in 1977 Pnueli[27] used temporal logic to prove properties of concurrent programs[10].

Literature differences temporal logic, depending on the interpretation of how system changes over time:

- **Linear temporal logic**

  Time is interpreted over a path through the model.

- **Branching temporal logic**

  Time is modelled as a tree with root in initial state. Tree is built through exploring all possible paths.

The literature typically presents CTL*, a logic with two notable subsets: Linear Temporal Logic(LTL) and Computational Tree Logic(CTL). Unlike in CTL, which is a *branching temporal logic*, in LTL temporal operators are being used to describe system's behaviour along one path in model. CTL and LTL share common temporal operators 'with different definition and semantics, but identical meaning(NeXt, Until, Future, Globally etc.). It is worth noticing, that CTL, LTL and CTL* may seem to be very similar, but they have different expressive power, i.e. there exists formulas in LTL that do not have equivalence in CTL and vice-versa. CTL*, as superset of both LTL and CTL, contains formulas that can not be expressed in LTL, or in CTL, or in any of this

[9] Terminology via [26].

[10] Pneuli's research on temporal logic had strong influence on verification of concurrent systems. He was rewarded with Turing Award in 1996, for "seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification."

logics[14].
We will introduce CTL for the purpose of next chapter, where two remarkable derivations of CTL will be discussed - Probabilistic CTL(PCTL) and Continuous Stochastic Logic(CSL), both used as a property language in simulator engine[11].

Generally, there exist various definitions of CTL in literature and they vary on existence of difference between state and path formula, symbols for temporal operators, appearance of derivable operators in syntax. To preserve compatibility with PCTL/CSL, due to their usage in project, we will follow convention of syntax with *state formula* and *path formula*, English capital letters as symbols of temporal operators and minimal set of operators in definition. This definition is compiled from [10][14].

Definition A CTL *state formula* over the set AP of atomic proposition is:

$$\Phi \ ::= \ true \mid a \mid \Phi_1 \ \wedge \ \Phi_2 \mid \neg\Phi \mid \boldsymbol{A}\,\phi \mid \boldsymbol{E}\,\phi$$

where $a \in AP$. A CTL *path formula* is:

$$\phi \ ::= \ \boldsymbol{X}\,\Phi \mid \Phi_1\,\boldsymbol{U}\,\Phi_2$$

CTL formula is a *state formula*.

We will also provide additional logical operators:

1. $\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$
2. $\phi \implies \psi \equiv \neg\phi \vee \psi$
3. $\phi \iff \psi \equiv (\phi \implies \psi) \wedge (\psi \implies \phi)$

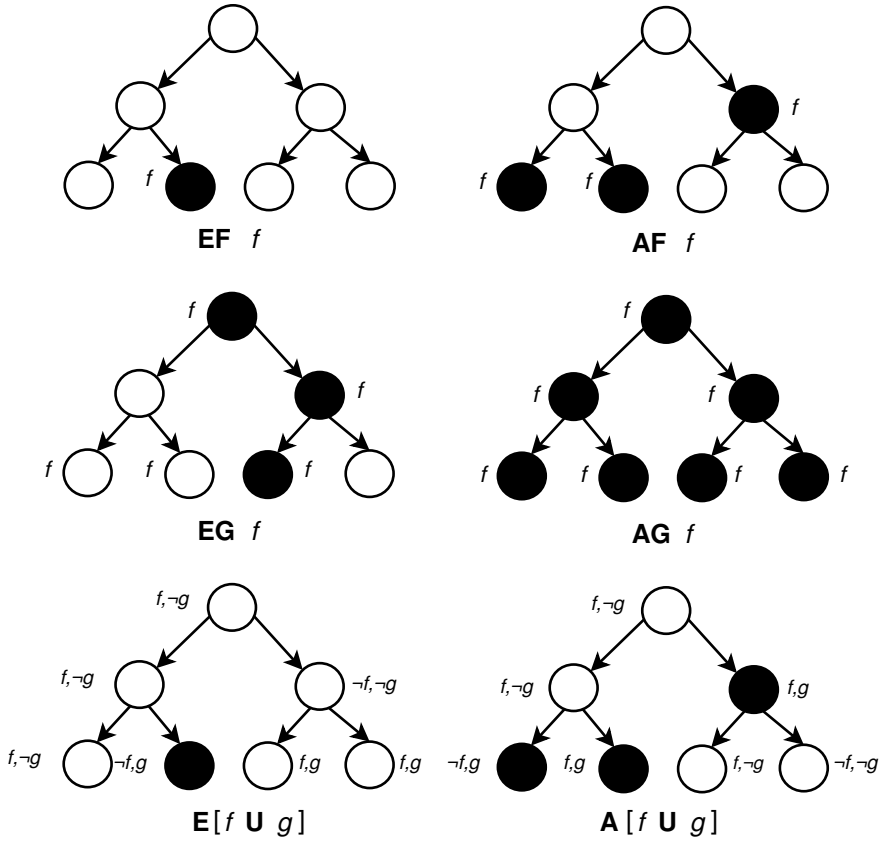
Definition clearly splits CTL formulas into pairs of two symbols. The first element in pair is path quantifier: **A** or **E**. A means "along All paths" and E means "there Exist at least one path". The second operator in pair is a temporal operator.
The CTL semantics will be defined on *states* and *paths* in the Kripke structure; symbol $\models$ stands for *satisfy*.

1. $s \vDash a \iff a \in L(s)$
2. $s \vDash \Phi_1 \wedge \Phi_2 \iff s \vDash \Phi_1 \wedge s \vDash \Phi_2$
3. $s \vDash \neg\Phi \iff s \nvDash \Phi$
4. $s \vDash \boldsymbol{A}\,\Phi \iff \forall_{\pi,\pi(0)=s_0}\ \pi \vDash \Phi$
5. $s \vDash \boldsymbol{E}\,\Phi \iff \exists_{\pi,\pi(0)=s_0}\ \pi \vDash \Phi$

The semantics of temporal operators is explained with the help of graphical interpretation in the Figure 2.2; filled circles represent a first state, in which property is satisfied.

[11] For description of LTL please refer to the literature, e.g. [10] or [3].

Figure 2.2: Basic CTL operators.



- **Next**

  Unary operator **X** specifies property which has to be satisfied by **next** state in path.

  $\pi \vDash \boldsymbol{X}\, \phi \iff \pi(1) \vDash \phi$

- **Until**

  Binary operator **U** requires that first formula remains valid **until** second formula becomes true.

  $\pi \vDash \phi\, \boldsymbol{U}\, \psi \iff \exists_{i \geq 0}\, \pi(i) \vDash \psi \wedge \forall_{0 \leq k < i}\, \pi(k) \vDash \phi$

- **Finally**[12]

  $\boldsymbol{F}\, \phi \equiv \boldsymbol{true}\, \boldsymbol{U}\, \phi$

  Unary operator **F** is used to signalize that property **eventually** is satisfied.

  $\pi \vDash \boldsymbol{F}\, \phi \iff \exists_{i \geq 0}\, \pi(i) \vDash \phi$

- **Globally**

  $\boldsymbol{G}\, \phi \equiv \neg(\boldsymbol{true}\, \boldsymbol{U}\, \neg\phi)$

[12] Also named "eventually", "in the future"

Property declared with unary operator **G** must always hold.

$\pi \vDash \boldsymbol{G}\,\phi \iff \forall_{i \geq 0}\, \pi(i) \vDash \phi$

- **Release**

  $\phi\, \boldsymbol{R}\, \psi \equiv \neg(\neg\phi\, \boldsymbol{U}\, \neg\psi)$

  Binary operator **R** indicates, that ψ is true until ϕ becomes true, including that state, or ψ is always true.

  $\pi \vDash \phi\, \boldsymbol{R}\, \psi \iff (\exists_{i \geq 0}\, \pi(i) \vDash \phi \wedge \forall_{0 \leq j \leq i}\, \pi(j) \vDash \psi) \vee (\forall_{i \geq 0}\, \pi(i) \vDash \psi)$

- **Weak until**[13]

  $\phi\, \boldsymbol{W}\, \psi \equiv (\phi\, \boldsymbol{U}\, \psi) \vee \boldsymbol{G}\, \phi$

  Binary operator **W** is similar to normal until with difference that ψ does not have to be eventually true.

  $\pi \vDash \phi\, \boldsymbol{W}\, \psi \iff (\exists_{i \geq 0}\, \pi(i) \vDash \psi \wedge \forall_{0 \leq k < i}\, \pi(k) \vDash \phi) \vee (\forall_{i \geq 0}\, \pi(i) \vDash \phi)$

Typical problems addressed at verification are[28]:

- **Reachability**

  Reachability property states that a particular situation will appear in path.

  In CTL it is expressed as $\boldsymbol{EF}\, \phi$, e.g. 'critical section will be reached' $\boldsymbol{EF}\, crit\_section$.

- **Safety**

  Safety property asserts that some event never occurs.

  In CTL it is expressed as $\boldsymbol{AG}\, \neg\phi$, e.g. "main and backup systems never fail simultaneously" $\boldsymbol{AG}\, \neg(system\_fail \wedge backup\_fail)$.

- **Liveness**

  In contradiction to safety, liveness makes a requirement that, under definite conditions, some event will eventually occur.

  Liveness may be expressed differently in CTL, e.g. 'each request will be eventually granted' as $\boldsymbol{AG}\, (request \implies \boldsymbol{AF}\, granted)$. In $\phi\, \boldsymbol{U}\, \psi$, the requirement that ψ will eventually be true is a liveness property and ϕ always holding is an instance of safety property.

- **Fairness**

  Fairness property states that, under definite conditions, some event will occur infinitely often. Fairness can not be expressed directly in CTL syntax, but there is an extension called *fairness constraints* for model[26].

[13] Also named "unless"

There exists an important distinction in properties, which splits them into two sets: *qualitative*, which seek an answer for questions about existence, occurrence etc., and *quantitative* used to describe measurable parameters of system. Quantitative properties may be verified in concurrent systems, but they are much more important in *probabilistic model checking*, described in next subchapter. Typical qualitative properties are: "will system reach an error state?", "every request will be granted eventually" or "if a message is sent, will it be delivered within 10 steps?".

Subsection 2.2.3 in next section presents and distinctly describes PCTL and CSL, logics used by simulator engine.

Verification Method

Verification methods may be entirely distinct for different approaches and they can be gathered into three main classes: analytical, simulation-based and analytical approximate:

- **analytical**
  group of all methods which process model algorithmically, mainly by doing exhaustive state-space search [14]

- **simulation-based**
  techniques which reason about system's behaviour by conducting series of model explorations; they provide ability to approximate quantitative results without checking whole state-space and that gives an opportunity to conduct at least basic verification, when model is too large to be processed by even the most advanced symbolic model checker

- **analytical approximate**
  methods like diffusion approximation has been widely applied to model and evaluate performance of queuing networks, analysing vast group of network properties(time, reliability, traffic control) [29].
  The basis of diffusion approximation is the replacement of discrete process(e.g. representing number of clients in serving station) with continuous diffusion process and its probability density function $f(x,t;x_0)$ is obtained by solving the diffusion equation[15]:

$$\frac{\partial f(x,t;x_0)}{\partial t} = \frac{\alpha}{2}\frac{\partial^2 f(x,t;x_0)}{\partial x^2} - \beta\frac{\partial f(x,t;x_0)}{\partial x} \tag{2.1}$$

[14]This report is focused on simulation techniques, so analytical algorithms will not be presented. Interested reader may find their description in [14][3], and for probabilistic model checking in [3, chap.10] or in lectures from course *Probabilistic Model Checking*, conducted by Dave Parker at the University of Oxford[7]

[15]the coefficients α and β eventuate from incremental changes of diffusion process $dX(t) = X(t+dt) - X(t)$, which is normally distributed and α, β are bonded to deviation and mean of this distribution

2.1.3 Probabilistic model checking

There is a great deal about verification of systems which exhibit probabilistic behaviour. Starting from pure ALOHA, through CSMA, CSMA/CD[16], CSMA/CA[17], and so forth, data link layer protocols have been using randomised choice of waiting time to resolve collisions, because clients will wait for a different amount of time[18] before next attempt to send a packet[30]; the *randomized leader election* is used in many standards of protocols and interfaces, e.g. the root contention procedure in IEEE 1394 FireWire. Moreover, probability is used to describe unpredictable behaviour in systems where some errors, problems or events may occur with estimated probability, e.g. packet loss in networks.

Probabilistic model checking[19] is a modification of standard approach in model checking, designed for analysis and verification of stochastic systems[31]. Model and property types have to be redefined, to contain additional information about the probability. Their extension allows to not only establish if a specific event may take place, but also to compute likelihood of its occurrence. Properties may contain limits on timing or risk of failure, e.g., 'the high priority packet will be delivered within 10ms with probability at least 0.99', 'with probability greater than 0.95 request will be granted within 1s' or 'what is the probability that fatal error will occur in system?'.

Probabilistic model checking has a wide range of applications: network and communication protocols[32], security[33], game theory problems, biological processes[34] and more. PRISM's website[35] contains an extensive list of case studies.

According to H.A. Oldenkamp's MSc thesis[36], who has done a comparative study of probabilistic model checkers, notable model checkers besides PRISM are: $E \vdash MC^2$[37], MRMC[38] and statistical model checkers YMER and VESTA, described in the next chapter.

2.1.3.1 Models for probabilistic model checking

The Kripke structure introduced in the previous section is no longer sufficient, when the task of checking state reachability is enhanced with computing the likelihood for the occurrence of particular events. Structure built from states and transitions contains information about all possible choice, but does not give an insight into probabilistic distribution of possibilities. Therefore we will use a different kind of automata and list of model types, commonly used by probabilistic model checkers, includes inter alia:

- Discrete-Time Markov chain(DTMC)
- Continuous-Time Markov chain(CTMC)

[16] The basis of IEEE 802.3, i.e., Ethernet LAN

[17] The basis of IEEE 802.11, i.e., Wireless LAN

[18] Very rarely there will be clash again and clients will random waiting time till success; the number of tries or delayment caused by them is a good example of property to verify.

[19] Also known as *stochastic model checking*

- Markov decision process(MDP)
- Probabilistic timed automata(PTA)
- Stochastic Petri nets

For the purpose of implementing new simulator engine, we will focus on Markov chains, named after famous russian mathematician Andrey Markov. Unlike most of the textbooks, mainly mathematical, where Markov chain is usually defined as a sequence of random variables[39], it will be treated by us as a Kripke structure with defined probability distribution for outgoing transitions from state. We believe, that for the purpose of model checking, state-transition view is more intuitive and natural.
We will consider two types of automata:

1. *discrete-time Markov chain*, which represents time as a sum of discrete steps and assign probabilities to each transition
2. *continuous-time Markov chain*, where time spent in a state is a real value and each transition contains its own rate

More general version of Markov chain is Markov decision process(MDP), which enables incorporation of probabilistic and nondeterministic behaviour. MDP contains set of probability distributions for each state and before randomizing next state in update procedure, one distribution is selected nondeterministically. For example, in distributed systems the concurrent updates of modules are nondeterministic rather than probabilistic, therefore they are typically modelled with MDP. Formally, Markov chain may be interpreted just as a MDP, but with only one distribution in each state.[20]

Before introducing formal definitions of DTMC and CTMC, we will present their common attributes:

- **graphical interpretation**
  the graphical view of Markov chain is exactly the same as in the Kripke structure, the only difference is the addition of probability/rate to each edge.
- **memorylessness**
  memorylessness, also known as Markov property, is an important property which distincts Markov chains from other models; they do not contain any information about states visited in the past, so this knowledge will not influence future choices. Next step in the model depends only on the current state.

[20]Currently simulation-based techniques are rarely use for *solving* nondeterminism. It is the main reason to focus on DTMC/CTMC in simulator.

- **time homogeneity**

  in this introduction we will focus on time-homogeneous Markov chains, i.e. the probabilities/rates of transition remain constant. Time-heterogeneous models are useful for situations where environment has an impact on strategy of choice. Simulator engine will be able to process this kind of Markov chains.[21]

Discrete-Time Markov chain

The definition is analogical with Kripke structure definition[31]:

Definition Let AP be set of atomic propositions. A *labelled DTMC* D over AP is a four tuple $D = (S, s_0, P, L)$ where

1. S is a finite set of states.
2. $s_0 \in S$ is the initial state.
3. $P : S \times S \to [0, 1]$ is the transition probability matrix where $\forall_{s \in S} \sum_{s' \in S} P(s, s') = 1$.
4. $L : S \to 2^{AP}$ is is a labelling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state

DTMCs take actions at times equal to subsequent natural numbers. System enters state at time n and performs transition to the next state at time $n + 1$, therefore, it stays there for one unit of time. Transition is randomized with the probability matrix, which contains probability distributions of transitions for each state. Obviously, transition is possible only if the probability of moving to the state s' is nonzero, i.e. $P(s, s') > 0$. The definition allows self-loops and the system may spend in state more than one unit of time, but it will be interpreted as a sum of single individual stays.

Continuous-Time Markov chain

Compared with DTMC definition, only transition matrix has different form[31]:

Definition Let AP be set of atomic propositions. A *labelled CTMC* C over AP is a four tuple $C = (S, s_0, R, L)$ where

1. S is a finite set of states.
2. $s_0 \in S$ is the initial state.
3. $R : S \times S \to \mathbb{R}_{\geq 0}$ is the transition rate matrix.
4. $L : S \to 2^{AP}$ is is a labelling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state

[21] PRISM allows to specify transitions rates as a function of model variables.

CTMC allows modelling systems in which events occur continuously, in real time. Time spent in a state is distributed exponentially, with rate determined by rate of the state s:

Definition The *exit rate* for the state s is:

$$E(s) \ = \ \sum_{s' \in S} R(s, s')$$

States, in which $E(s) \ = \ 0$, are called *absorbing*.

This approach to timing made from CTMC quite popular base for modelling, e.g. of queueing networks.
Traditional approach, found in the literature, reasons about CTMC execution in terms of joined probability of moving to the state s' within t time units:

$$P(s, s', t) \ = \ \frac{R(s,s')}{E(s)}(1 \ - \ e^{-E(s)t})$$

However, when generation of random paths in a simulator is considered, it is beneficial to treat transition selection and time spent in state separately.

Correspondingly with DTMC, transition from state s to s' can only occur if $R(s, s') > 0$. The situation when state has more than one outgoing transition is called *race condition*[22]: transitions participate in a race, where the 'fastest' is one with the highest rate, i.e., it has the highest probability of being triggered within t time units. First transition which sets off is the 'winner' of race.
To resolve race condition during simulation, we will use the concept of *embedded DTMC*[31]:

Definition The embedded DTMC of a CTMC $C \ = \ (S, s_0, R, L)$ is the DTMC $emb(C) \ = \ (S, s_0, P^{emb(C)}, L)$ where for $s, s' \ \in \ S$:

$$P^{emb(C)}(s, s') \ = \ \begin{cases} \frac{R(s,s')}{E(s)} & if E(s) \neq 0 \\ 1 & if E(s) = 0 \ \wedge \ s = s' \\ 0 & otherwise \end{cases}$$

Embedded DTMC allows identical algorithmic solution for selecting next transition in both automata, so there is no need for the simulator to reason about drawing transition in terms of an exponential distribution.

2.1.3.2 Logics for probabilistic model checking

We will introduce two temporal logics: Probabilistic Computation Tree Logic(PCTL) and Continuous Stochastic Logic(CSL), designed for verification of DTMCs and CTMCs, respectively. PCTL and CSL are extensions of CTL presented in chapter 2.1.3 and we will discuss only additional or modified parts of this language.

[22]Similar to concept of race condition in electronics(logic circuits) or software(multithreading applications), where system's behaviour is affected by the order of concurrent events.

We have made an important distinction between *qualitative* and *quantitative* properties. In a probabilistic model checking this difference is more natural. *Qualitative* property requires that probability of formula being valid in a system is equal to 0('no') or 1('yes'), and *quantitative* properties will be defined on an interval of possible values of probability[40].

Probabilistic Computation Tree Logic

Firstly introduced by Hansson and Johnson[41], Probabilistic Computation Tree Logic(PCTL) adds to the CTL probabilistic operator P[31]:

Definition A PCTL *state formula* over the set AP of atomic proposition is:

$$\Phi \ ::= \ true \mid a \mid \Phi_1 \ \wedge \ \Phi_2 \mid \neg\Phi \mid P_{\sim p}[\phi]$$

A PCTL *path formula* is:

$$\phi \ ::= \ \boldsymbol{X}\ \Phi \mid \Phi_1\, \boldsymbol{U}^{\leq k}\, \Phi_2$$

where $a \in AP$, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0,1]$ and $k \in \mathbb{N} \cup \{\infty\}$. PCTL formula is a *state formula.*

State s in DTMC satisfies PCTL property $P_{\sim p}[\phi]$ when the probability measure[23] of all paths from state s, that satisfy ϕ, fits in interval defined by $\sim p$.
Second relevant addition is *bounded until* $\boldsymbol{U}^{\leq k}$. The semantics of 'classic' **U** operator is enhanced with timing restriction, which demands that property will be satisfied within k time untis. Original unbounded until is expressed as $\boldsymbol{U}^{\leq \infty}$.

Continuous Stochastic Logic

Developed by Aziz et al.[42] and Baier et al.[43], Continuous Stochastic Logic is an extension of CTL and PCTL, designed initially for verification of CTMCs. CSL extends time domain from natural numbers to non-negative real numbers and allows verification of properties in steady-state[24].
CSL syntax expresses different time interpretation, if compared with PCTL[31]:

Definition A CSL *state formula* over the set AP of atomic proposition is:

$$\Phi \ ::= \ true \mid a \mid \Phi_1 \ \wedge \ \Phi_2 \mid \neg\Phi \mid P_{\ p}[\phi] \mid S_{\ p}[\phi]$$

A PCTL *path formula* is:

$$\phi \ ::= \ \boldsymbol{X}\ \Phi \mid \Phi_1\, \boldsymbol{U}^{\boldsymbol{I}}\, \Phi_2$$

where $a \in AP$, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0,1]$ and I is an interval of $\mathbb{R}_{\geq 0}$. PCTL formula is a *state formula.*

[23] For simplicity of this introduction, we have discounted the topic of probability measurement when DTMC was defined. For details please refer to literature, e.g. [31]
[24] Steady-state property is intended for verification of system in a 'long run', where equilibrium state is reached.

The semantics of P operator is the same as in PCTL. New temporal operator S indicates verification of formula in the steady-state. The 'until' operator is defined differently, but interpretation is intuitive: U^I is satisfied if and only if Φ_2 is true at some point in time in interval I and Φ_1 has not become false earlier.
CSL may be used for transient[25] probabilities, it is simply defined as $U^{[t,t]}$. The CTL's 'unbounded until' changes in CSL to $U^{[0,\infty]}$.

2.1.4 Statistical probabilistic model checking

As well as the model checking, its extension focused on stochastic systems suffers too from the problem of state-space explosion. Numerical methods offer a high degree of accuracy, but currently they can handle state space up to 10^7 states. Application of *Monte Carlo method*, an approach commonly used in computer science, to probabilistic verification results in a technique known as the *statistical probabilistic model checking*. Single exploration of whole state-space is replaced with repeated *sampling*, where state-space is generated 'on-the-fly', during each simulation. However, the benefits come with the cost: computed probability may be close to the actual value, but its accuracy is worse than one coming from numerical techniques. Fortunately, we are able to bound the *approximation error* and adjust it with calculating necessary sampling parameters.
Figure 2.3 presents the process of statistical approach to probabilistic model checking, according to [44]. This project is focused on new implementation for the first two steps: *path generation* and *property verification*; the *statistical analysis* in PRISM will not be changed, so the methods for statistical analysis are described without details needless for presentation of simulator engine.

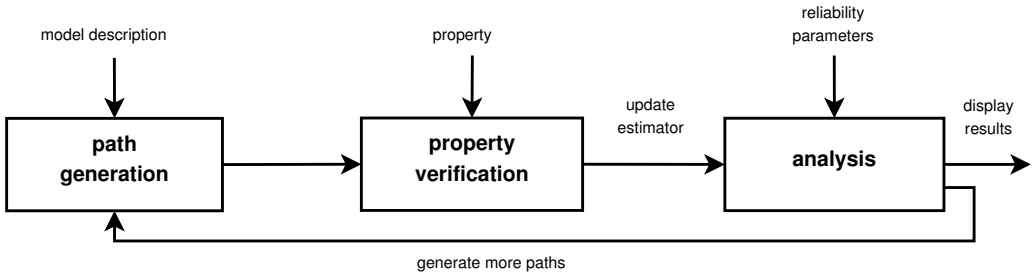


Figure 2.3: Overview of statistical model checking.

Firstly, the method for random path generation is presented. It is a modification of algorithms found in [45] and [46], in which we replaced all mathematical formalism with strictly programming view. Version for CTMC uses indirectly the concept of *embedded DTMC*:

[25]Transient property is intended for verification of system at a definite time instant.

Algorithm 1 A path generation algorithm for CTMC

1: init *state*, $\tau \leftarrow 0$
2: **while** maximal path length is reached[26] **do**
3: calculate transition rates $\lambda_1, \lambda_2, \ldots, \lambda_n$
4: calculate the sum of all transition rates $\lambda = \sum_{i=1} n\lambda_i$
5: draw an uniformly random variables x_1, x_2
6: find transition j for which $\sum_{i=1}^{j} \lambda_i \leq x_1 < \sum_{i=1}^{j+1} \lambda_i$
7: update system with j-th transition
8: update time $\tau = \tau - \frac{ln(1-x_2)}{\lambda}$
9: **end while**

Because time spent in a state is exponentially distributed with rate λ and we draw a uniform random variable, we use the *quantile function*[27] of an exponential distribution:

$$F^{-1}(p;\lambda) = \frac{-ln(1-p)}{\lambda}, \quad 0 \leq p < 1 \tag{2.2}$$

where ln is a natural logarithm.

Next algorithm presents a simplified version for path generation in DTMC:

Algorithm 2 A path generation algorithm for DTMC

init *state*, $\tau \leftarrow 0$
2: **while** maximal path length is reached **do**
calculate transition probabilities $p_1, p_2, \ldots, p_n$
4: draw an uniformly distributed random variable x_1
find transition j for which $\sum_{i=1}^{j} p_i \leq x_1 < \sum_{i=1}^{j+1} p_i$
6: update system with j-th transition
update time $\tau = \tau + 1$
8: **end while**

Several statistical techniques have been developed to verify if number of generated samples is sufficient, regarding expected accuracy of results. Properties expressed with the **P** operator can take two different forms: $\boldsymbol{P}_{=?}$, where we want to directly know the estimated probability, or $\boldsymbol{P}_{\sim\alpha}$, in which operator is bounded and we want only a Boolean answer if computed probability fits into given interval.

For properties from the first group three methods may be used: *CI* based on confidence interval, *ACI* using asymptotic confidence interval or *AMC*. The second type of property transforms easily to the first type, needing only one step after all computation to check if constraint holds. However, with this approach we may found ourselves in situation where the estimation error is too large, condition cannot be evaluated and we have to repeat sampling with stronger reliability parameters. This problem could be avoided with performing statistical test on hypotheses and for this purpose the *SPRT* method is provided.

[26] Break condition may be replaced with time constraint $\tau \leq \tau_{stop}$.
[27] Inverse of cumulative distribution function

As it was mentioned at the beginning of this section, we present only a short recap of an extensive review of this problem in [44].

Confidence Interval

During the sampling process, each path is checked whether property holds within, and as a result we obtain a sequence of Bernoulli random variables. Dividing their sum by the number of samples N gives requested approximation Y, but we do not know how far it is from the real value X. To create a bound on this error, the *confidence interval* is used. Defined by two parameters: width w and confidence level α, it expresses a guarantee that the value, that we're trying to estimate, will fit into this interval with probability $1 - \alpha$:

$$CI = [Y - w, Y + w] \tag{2.3}$$

In this method confidence interval is defined with the Student's t-distribution with N - 1 degrees of freedom:

$$CI = \left[Y - t_{N-1,1-\alpha}\sqrt{\frac{S^2}{N}}, Y + t_{N-1,1-\alpha}\sqrt{\frac{S^2}{N}} \right] \tag{2.4}$$

where S^2 is an estimation of the variance for X, and $t_{N-1,1-\alpha}$ is the quantile function of Student's t-distribution with N - 1 degrees of freedom.

The condition on the of width of interval joins three parameters of this method: width, confidence level and number of samples. Most frequently, we want to compute sufficient number of samples, which must hold condition mentioned below:

$$N \geq \frac{t_{N-1,1-\alpha} S^2}{w^2} \tag{2.5}$$

For situations when we know the number of samples and we're looking to compute width or confidence level, this equation may be used as well; the transformation for w is trivial, only extraction of α requires introducing the cumulative distributive function of the Student's t-distribution, denoted as $CDFt$:

$$\alpha = 2\left(1 - CDFt_{N-1}\left(w\sqrt{\frac{N}{S^2}}\right)\right) \tag{2.6}$$

Asymptotic Confidence Interval

The *asymptotic confidence interval* method is similar to previous technique, with the only difference that Normal distribution is used instead of Student's t-distribution. The final bound for N has to change, because quantile function of Student's distribution is replaced with its equivalent from Normal distribution:

$$N \geq \frac{z_{N-1,1-\alpha} S^2}{w^2} \tag{2.7}$$

The update in equation for w is identical, only formula for α requires putting the cumulative distributive function of Normal distribution $CDFn$ as a substitute of $CDFt$.

Approximate Model Checking

As it was proposed in [47], the construction of the confidence interval may be replaced with solving inequality:

$$Prob(|Y - X| \leq \epsilon) \geq 1 - \delta \tag{2.8}$$

where ϵ and δ are new parameters, called 'approximation' and 'confidence', respectively. However, the interpretation is similar to CI/ACI problem: probability that difference between approximated and real values is less than fixed limit, is equal to $1 - \delta$ or greater.
Solution is based on the Chernoff-Hoefdding bound:

$$Prob(|Y - X| \leq \epsilon) < 2e^{-2N\epsilon^2} \tag{2.9}$$

and the lower bound for the number of samples N becomes:

$$N \geq \frac{ln(\frac{2}{\delta})}{2\epsilon^2} \tag{2.10}$$

Similarly to previous techniques, two parameters are sufficient and the third may be computed straight away. However, unlike CI/ACI methods, ACM puts a constraint on N and ϵ when δ is not specified:

$$N\epsilon^2 \geq \frac{ln2}{2}$$

According to analysis and comparison in [44], in terms of the required number of iterations ACM is the less optimal. Yet it is a valuable method, because all parameters can be computed before simulation, no updates after required after path evaluation and the number of samples does not change during simulation and that makes this technique the easiest to implement with GPU.
On the other hand, CI/ACI methods require fewer samples than ACM, but after processing each path they will have to check if the bound is reached. Unlike the ACM, number of samples cannot be computed before simulation(if it was not specified) and that makes it a little more difficult to use when sampling is done by GPU. CI is the most expensive, due to recomputation of quantile after each sample. The disadvantage of ACI is that the computed confidence interval may be bigger than the one required.

Sequential Probability Ratio Test

When property has bounded probability, an alternative approach to standard techniques is to perform a statistical test.
The SPRT method performs the Wald's *sequential probability ratio test* to determine which one of the hypotheses is true: H_0 equal to property being valid and its adverse H_1, with the objection that probability value from property θ is replaced with its *indifference region* $[\theta - \delta, \theta + \delta]$, and δ

known as 'indifference is the first parameter. SPRT requires also two values to be specified: α as 'Probability of type I error'[28] and β as 'Probability of type II error'[29].

SPRT's advantage over other methods relies in guarantee, that after simulation we will be able to produce reliable answer. The number of samples is not known before simulation, so with GPU an approach similar to CI/ACI is required.

[28]Type I error is wrong acceptation of first hypothesis.
[29]Type II error is wrong acceptation of second hypothesis.

2.2 General-purpose computing on GPU

For many decades, the performance of computations has been increasing by manufacturing new generations of processors. Architectural improvements and growing density of transistors on die[30] has been holding up technological advance, but this situation rapidly changed in last few years. As an answer to technical problems caused by increased power consumption and heat development, multi-core processors has been presented. This fundamental change was great for operating systems, because they have been using concurrent threads for many years. On the other hand, the programmers had to learn how to find these parts of problem, that may be executed concurrently, and how express them in terms of programming environment. The range of complexity of parallel problems is large: from performing similar operations on each pixel, to sophisticated and synchronized streams of computations.

The easiest and the most natural to parallelize were *data parallelism* problems, where identical (or at least very similar) sequence of operations is executed on a huge amount of data[31]. Hardware solutions has been developed since 1970s and they became a standard when vector extensions[32] were introduced to desktop computers. SIMD intructions are widespread in 3D graphics operations, therefore it had strong influence on the architecture of Graphical Processors Units(GPUs).
In the last few years, the *general-purpose computing on graphical processor units* has become one of the most important concepts in parallel programming. The ability to perform on GPUa wide range of computation, not only those connected to graphical operations, allowed for large increase in performance of many algorithms and problems. The GPU architecture, focused on *stream processing*, i.e. processing one set of operations on many fragments of data in parallel, has several restrictions which makes from its usage useful only for some types of problems.

We have observed, that GPUs are good environment for the task of generating many samples, the basis of statistical model checking. Each path through model is independent and the usage of the biggest bottleneck - memory - may be minimized.
For this project we have chosen **OpenCL**, an open computing standard for *heterogenous*[33] platforms.

2.2.1 OpenCL

OpenCL is designed to allow single program to execute on many various platforms, which makes it powerful, but sometimes complicated, tool. Each hardware, compatible with OpenCL, comes with a set of tools for building and testing applications. Developer stays independent from the vendors,

[30] Observed by Gordon Moore, known as the 'Moore's law'
[31] From Flynn's taxonomy we would choose, for this problem, computer architectures: **SIMD** - Single Instruction, Multiple Data, or **MIMD** - Multiple Instruction, Multiple Data.
[32] Intel's MMX/SSEx/AVX, AMD's 3DNow!, common FMA3/FMA4.
[33] Platforms consisting from CPUs, GPUs, dedicated FPGAs and other processors.

their specific SDKs and languages. Moreover, OpenCL provides possibility to execute a program on many devices simultaneously, whether it is CPU or GPU. With the first release in December of 2008, reviewed versions in 2010 and 2011, and 2.0 version finished by the end of 2013, it is still young and developing platform, but with strong support of industry: AMD supports OpenCL through its Accelerated Parallel Processing SDK, NVIDIA's CUDA supports execution of OpenCL programs on GPU, Intel CPUs and GPUs are supported as well[34].

OpenCL execution model makes a distinction between *host* and *devices*. Host is responsible for choosing and initialization of OpenCL platform, creation of memory resources and compilation of a program into an OpenCL *kernel*, which will be executed concurrently on devices. To properly explain difference between conventional programming platforms and OpenCL, the execution timeline will be described:

1. The OpenCL *context*, a collection of runtime objects, is created on the selected platform.
2. One or more devices, enabled on the platform, are assigned to context. For each device a *command-queue* is created.
3. Host may allocate memory objects on device. The most common type is a *memory buffer*, containing an array of values.
4. Source code is compiled, which generates kernels for each device. The procedure is the same for all types of devices, OpenCL implementation is responsible for producing appropriate bytecode.
5. The data may be copied from host memory to buffers allocated on devices; it will be used as input values for computations.
6. Kernel execution command is put into command-queue.
7. The kernel saves results into buffers and accessing these values requires copying data from device to host memory;

When queue starts execution of kernel, it creates a collection of *work-items*, each one being a single instance of kernel call, with its own instruction counter, private variables and unique *global ID*, therefore, it is not a simple SIMD because each work-item may diverge on branching statements in the code. NVIDIA introduced the concept of **SIMT**, the *Single Instruction, Multiple Threads* to underline that the work-item is more related to a thread than just a sequence of identical operations, like in vector instructions. The literature presents OpenCL programming model as well as the **SPMD**, the *Single Program Multiplied Data*[48].

[34]Currently Intel SDK for OpenCL does not support Intel Core processors on Linux, only Intel Xeon are maintained. Instead AMD APP SDK may be used, which supports the x64 Intel CPUs.

The global space of work-items is evenly divided into the *work-groups*, each one of the same dimension and size. To properly understand this approach, we use the schematic view of an OpenCL device on the Figure 2.4.

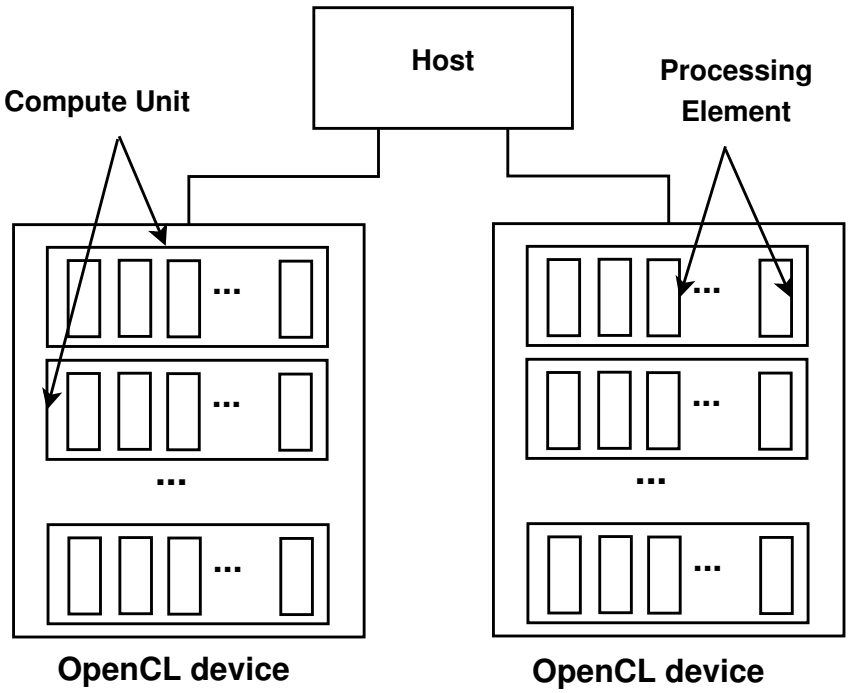


Figure 2.4: Schematic view of an OpenCL device.

The work-groups consists of work-items that execute concurrently on processing elements of one compute unit. Actually, this is the only one place where standard guarantees parallelism; the execution of work-groups may be completely serialized[48].
The global space is known as the *NDRange*, where N denotes its dimensionality[35]. Work-item's global ID and local ID, the unique ID in work-group, consists of N natural numbers. Therefore, a single thread may be identified in global space with global ID, in local space with local ID, and a work-group is identified with its own work-group ID.

2.2.1.1 OpenCL memory model

The memory in OpenCL is divided into four disjoint spaces, as shown on the Figure 2.5.

The reason of such memory organization is the universality of OpenCL, which has to perform efficiently on different hardware architectures. The abstract spaces, specified in standard, are mapped into hardware memory in implementation; the example of this implementation-dependent approach is presented in section 2.2.1.3, where the execution on GPU is described.

[35] Currently OpenCL supports 1,2 or 3-dimensional space of work-items.

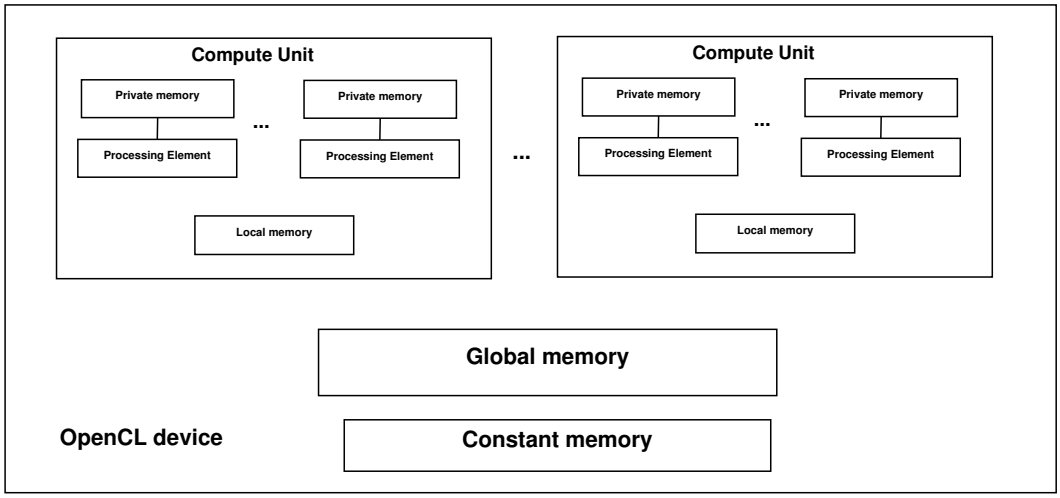


Figure 2.5: Schematic view of memory spaces in OpenCL.

The **private memory** consists of all variables in work-item that are not declared to be local or global. The values are stored in registers, so the latency for memory operations is negligible. Unfortunately, the number of registers available on processing unit is limited and the high usage of private memory in kernel may affect performance in two ways - by decreasing the maximal number of concurrent threads or by pushing register values into memory. On CPU, the cache may be used, which adds only a small waiting time. However, the GPUs often does not have cache on multiprocessors and the surplus registers will be moved to global memory, which may drastically decline performance.

The **local memory** may be accessed and modified by all work-items within one work-group. Local memory provides the possibility for work-items to communicate. Usually, this memory is physically located on the compute unit, which makes it much faster than global memory; the programmer is free to use local memory as a cache or as a slower, but typically larger private memory.

The **global memory** is accessed by all work-items on device. Its size and read/write latency depends on the type of device, but it is normally used for work-item input data and results. Performing frequent operations on global memory has a strong and negative impact on performance.

The **constant memory** is allocated by the host and work-items are not allowed to modify it.

The OpenCL uses a *relaxed consistency model*, which allows for non-consistent memory values across the NDRange. In the work-item, the order of memory operations is not changed. In the work-group, the work-items may be synchronized to obtain memory coherence. The *fence* command guarantees that all write/read operations to local or global memory will complete, before any such operation starts after the fence. In addition to fence, the *barrier* blocks threads until each work-item in work-group reach this instruction[49].

2.2.1.2 OpenCL C programming language

For programming kernel, the *OpenCL C* is used[50]. It is an extension of the C programming languages, with the several restrictions:

- Functions defined with keyword __kernel are the only one that the host may execute.
- All variables have to be defined in the method body. Global variables are not allowed.
- OpenCL extends standard scalar types in C by defining its size and adding new types[36]. Table shows the most common scalar data types:

OpenCL C type	OpenCL API type	Description
bool	n\a	True/false, casted to integer '1' and '0', respectively
char, unsigned char	cl_char, cl_uchar	Signed/unsigned 8-bit integer
short, unsigned short	cl_short,cl_ushort	Signed/unsigned 16-bit integer
int, unsigned int	cl_int,cl_uint	Signed/unsigned 32-bit integer
long, unsigned long	cl_long,cl_ulong	Signed/unsigned 64-bit integer
float	cl_float	A 32-bit floating-point, conforms to the IEEE 754 single precision.
double	cl_double	A 64-bit floating-point, conforms to the IEEE 754 double precision.
half	cl_half	A 16-bit floating-point, conforms to the IEEE 754-2008 half precision.
void	void	Identical with void type in C programming language.

  The boolean variable exist only in a kernel source, it can not be used as a kernel argument. The double scalar is not supported on all devices, it may be checked on runtime with preprocessing conditionals.

  OpenCL C provides also vector data types for all integer types, float and double. The permitted sizes of a vector are: 2,3,4,8 and 16.

  For accessing different memory spaces, OpenCL C has three qualifiers: __global,__local and __constant. When no qualifier is given, the default space is private

- The usage of pointers is restricted. When pointer is an argument to kernel function, it must point to global, local or constant space. The pointers declared with different memory qualifiers cannot be assigned. Function pointers are not allowed.
- Only some of the C libraries are available, e.g. math.h.
- Recursion is not permitted.

[36]The C standard does not define the built-in scalar size, only describes the size relation between different types.

The Listing 2.1 presents an example of OpenCL kernel - vector addition on one-dimensional space of work-items.

```
__kernel void main(__global const float * A, __global const float * B, __global float
    * C) {
    uint globalID = get_global_id(0);
    float result = A[globalID] + B[globalID];
    C[globalID] = result;
}
```

Listing 2.1: An OpenCL kernel vecAdd.cl

2.2.1.3 OpenCL on GPU

This section presents execution model and memory mapping on NVIDIA CUDA architecture, according to [51][52]. We use this brief description to underline the biggest problems, possible bottlenecks and differences with the CPU. For the AMD graphic processors, the idea remains the same; more detailed description may be found in [53].

A CUDA device consists of multithreaded *Streaming Multiprocessors*, which corresponds to the computing unit from OpenCL architecture. The work-groups, known in the CUDA as *thread blocks*, are distributed among multiprocessors, where work-items are executed in groups of 32 threads called *warps*. All threads in warp can execute only one instruction at the time, so if some threads diverge due to conditional branch, the warp will serialize execution, computing each path taken by thread. This problem has to be considered meticulously by the programmer, because it may seriously decrease performance.

All NVIDIA GPUs are able to process at least 768 active threads per multiprocessor. The warp scheduler may switch between active warps to hide latency, caused by e.g. access to global memory. This procedure does not generate overhead, because registers are allocated to warp and no swapping is required.

The OpenCL local memory is mapped to CUDA shared memory, located on multiprocessor. The multiprocessor resources consists of 8,192, 16,384 or 32,768 32-bit registers and 16 or 48 kB of shared memory, depending on device *Compute Capability*.

The practices, that are most important for high performance, are:

1. Parallelize code in kernels, to avoid the divergence in execution paths within one warp.
2. Minimize data transfer between the host and devices.
3. Use local memory as a cache for transfers to global memory. If global memory is used, make the operations coalesced(if it is possible), i.e. fit reads and writes into segments of 16 and 32 words. Device tries to perform fewer transactions between multiprocessor and global memory.

4. Local memory is divided into banks, which can be accessed concurrently; the operations on it in kernel should be designed to avoid bank conflicts, i.e. different work-items accessing same bank.

Chapter 3

Problem Analysis

Chapter Abstract

This chapter provides an analysis of the proposed solution for new simulator engine. First section describes the structure of PRISM model, used by the engine. Section 3.2 presents two main algorithms of the approximate probabilistic model checking with abrief look at property evaluation methods.
Section 3.3 gives a brief view on the existing simulator in PRISM, followed by a detailed list of requirements for designed simulator in Section 3.4. Chapter ends with the summary of other statistical model checkers.

3.1 Model Specification in PRISM

PRISM has its own high-level programming language, based on the Reactive Modules formalism of Alur and Henzinge[54]. Probabilistic models are defined as the parallel composition of independent modulesl; the state-space of a model is a product of state-spaces of all modules, with the addition of global variables. PRISM reads model file and builds from it a discrete-time Markov Chain (DTMC), continuous-time Markov Chain (CTMC), Markov decision process (MDP) or a Probabilistic Timed Automata (PTA).
In [55] PRISM model is expressed as a tuple **(M,G,R)** which, in our opinion, currently should be extended to a tuple **(M,G,F,L,R)**, where:

- **M** is a set of modules, running concurrently and interacting with others through global variables or synchronization labels. The *local state* of module m is determined by a set V_m of integer or Boolean variables.
- **G** is a set of global variables, each one being an integer or a Boolean.
- **F** is a set of formulas, used to avoid duplication of code. A formula consists of an identifier and an expression, and each occurrence of the identifier in a model description is replaced with

formula's expression. It is similar to the concept of a preprocessor *define* directive in the C programming language.

- **L** is a set of labels, each one containing a conjunction of logical expressions. Labels are used in properties for identification of particular set of states.
- **R** is a set of *reward constructs*.

Each module contains a set of *commands*, denoted as $Comm_m$[1], which describes the behaviour of module using pairs of a *guard* and a list of *updates*. Guard is a Boolean expression over the set of local and global variables, which determines if the command is active in given state. Update consists of a *rate* and a list of assignments to variables. Command in module m is allowed to update only variables from sets G and V_m.

Transitions in the model can occur in two ways: as *asynchronous* execution of command in one module, or as an execution of one command in two or more modules at the same time, called the *synchronized* transition. The set *Act* of *action labels* denotes which transitions can occur simultaneously. More than one command in a module may be assigned to a single label and the transition can be executed only when at least one command is active in each synchronized module. Therefore, in each state of model one module makes an independent transition or one transition labelled with action a is processed.
Next two sections explain interpretation of transitions system in DTMC and CTMC.

3.1.1 Discrete-time Markov Chain

As it was described in section Chapter 2.1.3.1, in DTMC each state is connected with a probability distribution of outgoing transitions. Updates are taken with discrete time steps.
Each command c contains guard with the list of logical conditions, separated by & or | symbols, equivalent to logical conjunction and disjunction. The symbol $\rightarrow$ is followed by the set of updates, $Updates_c$, separated with + symbol. Each update $u \in Updates_c$ is in the form $P_u : Assign_c$, where:

- **P** is the probability of that update. The probability expressions may consists of numerical literals, arithmetic expressions and variables values. In DTMC holds the condition that:

$$\forall_{c \in Commands_c} \sum_{u \in Updates_c} P_u = 1 \tag{3.1}$$

- The set of variables updates $Assign_c$, separated by & symbol. The identifier of assignment destination is written with ´ suffix, and the new value may involve an arithmetical operations and calls of simple mathematical functions, e.g. min, max, log, pow.

[1] The mathematical formalism and nomenclature is compatible with the on presented in [55]

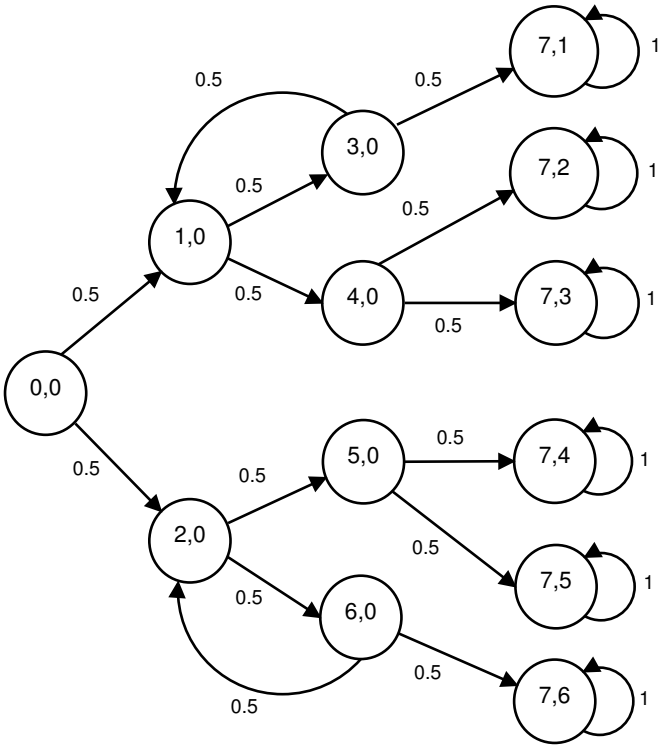
The Listing 3.1 presents an exemplary DTMC model in PRISM language. It is an algorithm for simulation of 6-sided die with a fair coin, given by Knuth and Yao[2][56].

```
dtmc

module die

  // local state
  s : [0..7] init 0;
  // value of the die
  d : [0..6] init 0;

  [] s=0 -> 0.5 : (s'=1) + 0.5 : (s'=2);
  [] s=1 -> 0.5 : (s'=3) + 0.5 : (s'=4);
  [] s=2 -> 0.5 : (s'=5) + 0.5 : (s'=6);
  [] s=3 -> 0.5 : (s'=1) + 0.5 : (s'=7) & (d'=1);
  [] s=4 -> 0.5 : (s'=7) & (d'=2) + 0.5 : (s'=7) & (d'=3);
  [] s=5 -> 0.5 : (s'=7) & (d'=4) + 0.5 : (s'=7) & (d'=5);
  [] s=6 -> 0.5 : (s'=2) + 0.5 : (s'=7) & (d'=6);
  [] s=7 -> (s'=7);
endmodule
```

Listing 3.1: PRISM model for die algorithm.

The Figure 3.1 presents the graphical interpretation of DTMC built from the above description.

Figure 3.1: DTMC built from model in Listing 3.1.



As it can be seen in the model, only one guard is active in each state. However, it does not have to be true in all models - the situation when more than one guard in the module is active in a

[2]Source of model: tutorial at PRISM page.

state, known as the *local nondeterminism*, is solved by randomized choice of a transition to execute. The behaviour is identical in a situation where there is more than one module with active guards. If N asynchronous commands are active along all modules, then each one will be processed with probability equal to $\frac{1}{N}$.
Because of that, each command in the model from Listing 3.1, with the exception to last one, is equivalent to two commands with the same guards and two updates split between them, with probability changed from 0.5 to 1.

The synchronized transitions are evaluated differently. For each action label a the update algorithm is identical:

1. For each module, evaluate guards in labelled commands.

2. Compute number of possible transitions s in the following way: for each module that contains at least one command with label a, compute number of active guards, denoted by $guards_{a,m}$. Then the number of synchronized transitions will be equal to:

$$s = \prod_{m \in M} guards_{a,m} \tag{3.2}$$

3. If m is not equal to 0, then create a new set of commands by computing all possible products, taking one command from each module. The product of commands c_1 and c_2 is defined as follows:

   - The action labels in c_1 and c_2 should be the same, so they determine the label of $c_1 \times c_2$.
   - $Guard_{c_1 \times c_2}$ is the logical conjunction of guards in c_1 and c_2, which have been evaluated earlier, so it will be not used.
   - The list of updates $Update_{c_1 \times c_2}$ is the product of $Update_{c_1}$ and $Update_{c_2}$, so for updates u_1 and u_2 their product $u_1 \times u_2$ consists of:

$$P_{u_1 \times u_2} = P_{u_1} \cdot P_{u_2} Assign_{u_1 \times u_2} = Assign_{u_1} \cup Assign_{u_2} \tag{3.3}$$

The number of synchronous transitions in model, denoted as S, is the sum of s transitions for each action label.

Therefore, for state with N active asynchronous guards and with S synchronous transitions, each update is selected with probability $\frac{1}{N+S}$.

3.1.2 Continuous-time Markov Chain

Continuous-time Markov Chain is a generalization of DTMC, where time spent in a state is no longer equal to one discrete step, but it is a floating point value. The probability P for transition

is replaced with the rate R, which denotes how 'fast' this transition will be triggered. The average time spent in a state before leaving is equal to $\frac{1}{R}$.

Contradictory to DTMC, the sum of rates for all transitions is not in the interval $[0, 1)$ and the situation where state has more than one outgoing transition is known as the race condition; it has been described in Chapter 2.1.3.1, along with the concept of embedded DTMC and computation of time spent in state.

The Listing 3.2 presents an exemplary CTMC model in PRISM language, the M/M/1 queue.

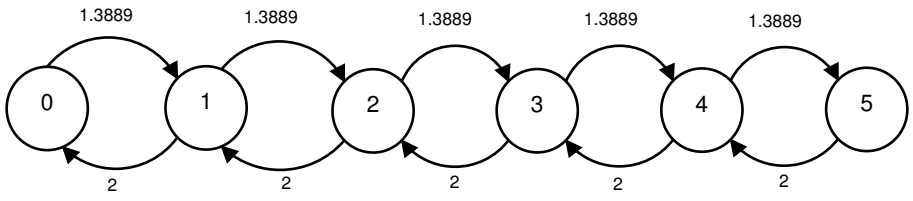
```
ctmc

//queue length
const int N = 5;
const double arrival_rate = 1/0.72;
const double service_rate = 1/0.5;
module mm1

  s : [0..N] init 0;

  [] s=0 -> arrival_rate:(s'=1);
  [] (s>0) & (s<N) -> arrival_rate:(s'=s+1) + service_rate(s'=s-1);
  [] s=N -> service_rate(s'=s-1);
endmodule
```

Listing 3.2: PRISM model for the M/M/1 queue.

The Figure 3.2 presents the graphical interpretation of an automaton built from the above description.

When more than one guard is active, the race condition occurs between all updates from each

Figure 3.2: CTMC built from model in Listing 3.2.



active command. Therefore, the second command from Listing 3.2 can be split into two commands, one for customer arriving to queue and one from customer leaving after service.

Processing synchronized transitions in CTMC is exactly the same as in DTMC. During path generation, the time update is the only difference between these two types of automata. For this purpose, as it was explained in Algorithm 1 in the previous chapter, we need to know the sum of rates of all outgoing transitions.

3.2 Approximate Probabilistic Model Checking

In [47] the *generic approximation algorithm* was presented. It was later extended in [55] by adding loop detection and considering multiple properties; we will present its modification which does not directly compute number of samples, but works with different approaches presented in Chapter 2.4.

The parameters of the algorithm are:

- M - The PRISM model
- $\phi[]$ - The array of properties
- k - The maximum path length

For simulation methods, that compute number of samples before simulation, we will consider one additional parameter:

- N - The number of samples

The algorithm for the first group of methods:

Algorithm 3 The Generic approximation algorithm for multiples properties[55]

1: **for each** $\phi \in \phi[]$ **do**
2: $A_\phi \leftarrow 0$
3: **end for**
4: **for** $i \leftarrow 0$ **to** N **do**
5: Start a path π in the initial state
6: $j \leftarrow 0$
7: $t \leftarrow 0$
8: **while** $j \leq N$ **do**
9: Compute leaving time t_1
10: **if** $\forall_{\phi \in \phi[]}\, known(\phi, \pi, j, t_0, t_1) = true$ **then**
11: End current path
12: **end if**
13: Randomize next state with M and add it to π
14: $j \leftarrow j + 1$
15: **if** $\pi(j) = \pi(j-1)$ & transition choice is deterministic **then**
16: Loop detected, end current path
17: **end if**
18: $t_0 \leftarrow t_1$
19: **end while**
20: **for each** $\phi \in \phi[]$ **do**
21: **if** ϕ is satisfied over π **then**
22: $A_\phi \leftarrow A_\phi + 1$
23: **end if**
24: **end for**
25: **end for**

The function $known(\phi, \pi, j, t_0, t_1)$ evaluates property ϕ over path π, with a current path length j, time of entering state t_0 and time of leaving state t_1. It is defined in the next section; for proper

understanding of the algorithm one has to know that this function returns a Boolean value, equal to true only if the value of property ϕ is known.

Other simulation methods provide for each property function $stop(A, n)$, where A is the number of paths on which property was positively evaluated and n is the number of processed paths. The modified algorithm for situations where number of samples is not known before simulation:

Algorithm 4 The Generic approximation algorithm for multiples properties without number of samples

```
1: for each φ ∈ φ[] do
2:     A_φ ← 0
3: end for
4: i ← 0
5: while ∃_{φ∈φ[]} stop(A_φ, n) = false do
6:     Start a path π in the initial state
7:     j ← 0
8:     t ← 0
9:     while j ≤ N do
10:        Compute leaving time t_1
11:        if ∀_{φ∈φ[]} known(φ, π, j, t_0, t_1) = true then
12:            End current path
13:        end if
14:        Randomize next state with M and add it to π
15:        j ← j + 1
16:        t_0 ← t_1
17:    end while
18:    for each φ ∈ φ[] do
19:        if φ is satisfied over π then
20:            A_φ ← A_φ + 1
21:        end if
22:    end for
23: end while
```

3.2.1 Suitable Properties for Simulation

[55] provides a detailed description of properties that can not or should not be simulated:

Unbounded Formulae

Properties with an upper time bound give the maximal path length in natural way - it is equivalent to time reaching the bound. However, in unbounded properties we reason about infinite paths which can not be generated by the simulator. For example, for correct approximation of property $P_{=?}\,[\boldsymbol{true}\,U\,\phi]$, the number of paths in which ϕ is satisfied has to be estimated. If there exist state s_j, in which ϕ becomes true, j is greater than the maximal path length k and ϕ is not satisfied for all states in the path of length k , then this path property will be incorrectly evaluated as false.

For solving this problem, the concept of *monotone* formula was introduced in [47]. It is defined as formula where 'truth of formula at length k implies truth in the entire model'.

Therefore, the value k should be provided by the user which has to check whether the formula is monotone at length k.

Steady-State Formulae

Steady state formula requires computation of steady state in a model, which is rarely done by a simulator - the result does not have to be completely correct and it is very inefficient due to huge length of path.

Nested-Path Formulae

PCTL/CSL syntax allows nesting path formula, e.g.

$$P_{=?}[\Phi_1 \, \boldsymbol{U} \; < \; 11.5 \, P > 0.5[\boldsymbol{true} \, \boldsymbol{U} \, \Phi_2]] \tag{3.4}$$

which means 'what is the probability that system will reach a state within 11.5 time u, from which the probability of reaching state which satisfies Φ_2 is greater than 0.5, and Φ_1 will be true in all previous states'.
The analysis of this properties in [55] leads to a conclusion, that different approaches for simulating this properties will be inefficient, in the context of time or memory consumption.

Multiple Initial States

PRISM allows properties specified with a set of initial states, denoted with a label. For exemplary label "initial":

$$'initial' \implies P_{=?}[\Phi_1 \, \boldsymbol{U} \, \Phi_2] \tag{3.5}$$

Because the approximation algorithm considers only path starting in one initial state, this kinds of properties are not evaluated by simulator

3.2.2 Property Evaluation

Properties are specified in PRISM in external files and their grammar is equivalent to original syntax, described in Chapter 2.1.3.2.

The evaluation of property during simulation is defined with the function $known(\phi, \pi, j, t_0, t_1)$, mentioned in the previous section. For each property, the function stores two Boolean variables, one containing its value and one containing information if the property has been already evaluated.

"Next"

- If the value is known, then return true.
- If $k = 0$, then return false.
- If $k = 1$, then evaluate the expression, save result and return true.

"Until"

- If the value is known, then return true.
- If right expression in property is satisfied, then save the result 'true' and return true.
- If left expression in property is not satisfied, then save the result 'false' and return true.
- Otherwise, the property can not be evaluated, return false.

"Bounded Until"

The approach for bounded operator **U** differs between model types.

In **discrete-time Markov Chain**:

- If value is known, then return true.
- If t_0 is greater or equal to upper time bound, then:
  - If right expression in property is satisfied, then save the result 'true' and return true.
  - Otherwise, save the result 'false' and return true.
- Otherwise, do the following:
  - If right expression in property is satisfied, then save the result 'true' and return true.
  - If left expression in property is not satisfied, then save the result 'false' and return true.
  - Otherwise, the property can not be evaluated, return false.

In **continuous-time Markov Chain**, the conditions for initial state appear:

- If lower time bound is greater than 0, then:
  - If left expression in property is not satisfied, then save the result 'false'.
- Otherwise:
  - If right expression in property is satisfied, then save the result 'true'.

The method is slightly different:

- If the value is known, then return true.
- If t_1 is greater than upper time bound, then:
  - If right expression in property is satisfied, then save the result 'true' and return true.
  - Otherwise, save the result "false" and return true.
- If t_1 is lower or equal than lower time bound, then:

– If left expression in property is not satisfied, then save the result 'false' and return true.

– Otherwise, the property can not be evaluated, return false.

Otherwise, do the following:

– If right expression in property is satisfied, then save the result 'true' and return true.

– If left expression in property is not satisfied, then save the result 'false' and return true.

– Otherwise, the property can not be evaluated, return false.

3.3 Current Simulator Engine in PRISM

Currently, PRISM contains a simulator engine, implemented as software project at the University of Birmingham[55], with later extensions, including different simulation methods in [44].

The features of the engine:

- The simulator engine works on parsed PRISM model.
- The engine contains the current state of a model.
- The engine provides methods for creating and manipulating an execution path.
- The engine allows to make any number of automatic updates.
- The engine allows approximate model checking of properties, using one simulation method, selected by the user.

Approximate model checking is possible in model types: DTMC, CTMC and MDP, and engine handles properties with operator **P** and **R**[3], without nested temporal operators.

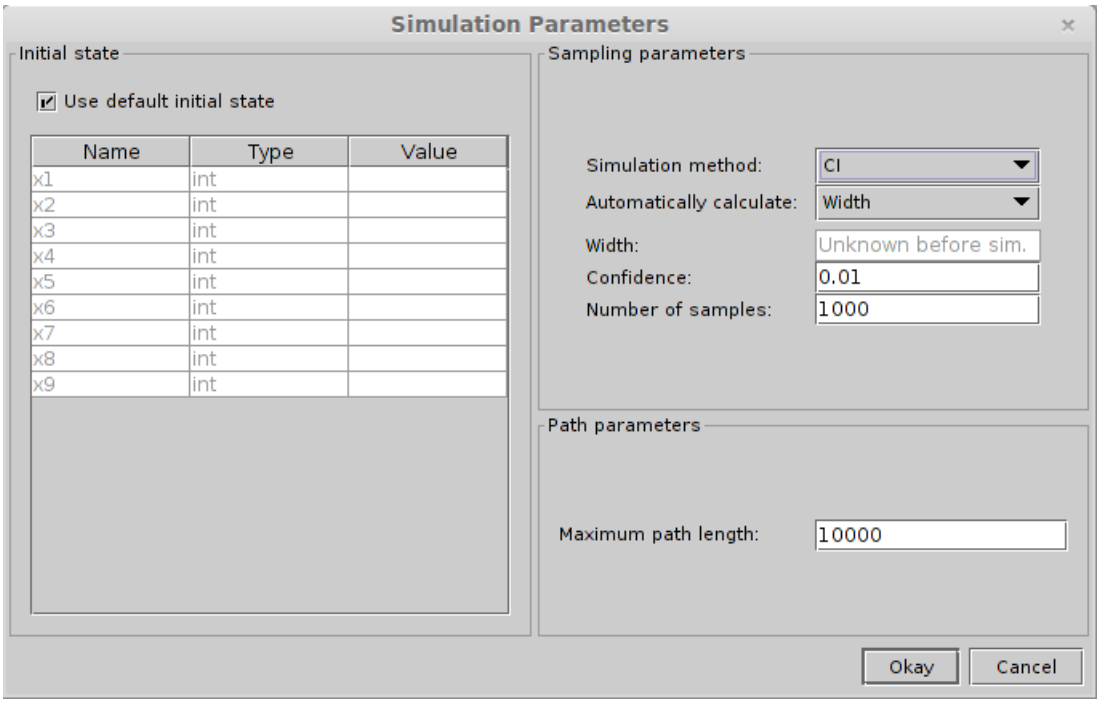
The engine is integrated with PRISM in two ways, through the Command-Line Interface(CLI) and the Graphical User Interface(GUI).

The GUI provides a possibility to execute path in model, manually or in partially automatic way. It is designed as a debugger for models. In addition, the approximate model checking can be processed using this interface.

Figure 3.3 presents a view of the window with parameters for stochastic model checking. The panel *Initial state* on the left allows for manual modification of an initial state, the panel *Sampling parameters* is used to select simulation method and specify its parameters.

[3] Additional reward operator.

Figure 3.3: Simulation panel in PRISM GUI.



3.4 Requirements for New Simulator Engine

The aim of this dissertation is to provide new, additional implementation of simulator engine, which will offer better performance on the task of approximate model checking. The disadvantage of current engine is the sequential processing of paths and computational overhead during generation of transitions for each state.

This section provides requirements for the new engine, starting from internal requirements and implementation architecture. Then, the possibilities of model checking are described. This sections covers also the requirements for user interface.

3.4.1 Structural Requirements

1. The engine will be included into existing architecture as a substitute for approximate verification. Therefore, the interface and the structure of results should be identical with the current simulation engine.

2. The engine will replace only stages of path generation and property verification in the process of statistical model checking, according to schematic view in Figure 2.3. Therefore, simulator will be using existing implementations of classes: Sampler and SimulationMethod[4].

3. The internal structure of the engine will consists of two parts: model preprocessor and runtime.

[4]Described in next chapter.

(a) The model preprocessor will extract an automaton from ModulesFile[5] object, extending formulas and constants.

(b) The model preprocessor will extract properties from PropertyFile[6] object, using existing Sampler[7] implementations.

(c) The model preprocessor will be suitable for the future implementations of pre-simulation optimization techniques.

(d) The runtime will provide an interface for simulation and setting additional user-defined parameters: initial state and maximal path length.

(e) The runtime will provide an interface for selection and exploration of possible hardware devices, that will be used for simulation.

4. The engine will contain one, main internal implementation in OpenCL.

3.4.2 Verification Requirements

1. The engine will process model types: DTMC and CTMC.
2. The engine will allow for verification of PCTL and CSL properties, accepted by PRISM, with the exception of property types mentioned in Section 3.2.1.
3. The engine will be using simulation methods implemented in PRISM, described in Section 2.1.4.
4. The path generation will be processed in parallel.
5. The path generation will be possible to execute on GPU, on devices supporting OpenCL execution.

3.4.3 User Interface Requirements

1. The engine will be integrated with existing user interfaces, making minimal number of changes in that field.
2. The verification will be possible through the Command-Line Interface, with the additional selection of simulator engine.
3. The verification will be possible through the Graphical-User Interface, with the selection of:

   - **Simulation Engine**

     Between existing engine and runtime frameworks implemented in the new one.

[5] Class in PRISM code that keeps contents of module file.
[6] Class in PRISM code that keeps contents of property file.
[7] Class in PRISM code that provides implementation of property evaluation and connects it with SimulationMethod object, responsible for the statistical computations.

- **Device Selection**

  Between enabled devices in the system, from each vendor and OpenCL platform.

The panel with selection will show detailed information about each device.

3.5 Related work

This section provides brief descriptions of another existing implementations of different approaches to statistical probabilistic model checking.

3.5.1 The Approximate Probabilistic Model Checker

The Approximate Probabilistic Model Checker(APMC)[57] is an approximate distributed model checker for stochastic systems. APMC uses the Generic Approximation Algorithm, presented in Chapter 3.2. APMC includes support for distributed generation of samples on a cluster of workstations. It is released under the GNU General Public Licence (GPL).
The APMC approximates the probability of a LTL formula being true over a DTMC or CTMC. Currently, the version 2 is no longer developed, and the version 3 written in Java is considered as a beta version.
APMC can be obtained from: http://sylvain.berbiqui.org/apmc

3.5.2 Ymer

Ymer[58] is an open-source tool for verification of probabilistic transient properties in CTMCs and Generalized Semi-Markov Processes(GSMPs). The accepted properties are written in the CSL logic. Tool is developed by Håkan L. S. Younes and it is released under the GNU General Public Licence (GPL). Ymer uses a subset of PRISM language for specifying models.
Ymer supports distributed generation of samples and the beta version 4 supports unbounded until, but does not contain all features from previous versions. In contradiction to PRISM simulator engine, Ymer uses the hybrid engine from PRISM to combine numerical and statistical approaches in the nested properties.
Ymer is available at: http://www.tempastic.org/ymer/

3.5.3 VeStA

VeStA[59] is a statistical probabilistic model checker, developed at the University of Illinois at Urbana-Champaign. VESTA was designed to verify PCTL, CSL and QuaTEx[8] properties on DTMCs and CTMCs. Currently, there is no possibility to access VESTA, because the webpage is not available.
In the last few years two extensions of VESTA have been developed:

[8] Quantitative Temporal Expressions

- PVeStA[60], providing parallel sampling
- MultiVeStA, supporting many discrete event simulators; available at http://code.google.com/p/multivesta/

3.5.4 PRISM-U2B

PRISM-U2B is a modification of PRISM, developed with a new method for verification of unbounded PCTL properties in DTMC[61].
The tool is available at: http://fmg.cs.iastate.edu/project-pages/pmck/

Chapter 4

Design and Implementation

Chapter Abstract

This chapter provides a description of class hierarchy and structure. First section presents internal structure of existing simulator engine, followed by a depiction of the new engine. The modifications in CLI and GUI are presented in section 4.3.
Section 4.4 shows the KernelGenerator library, created for the purpose of generation source code in OpenCL.

4.1 Prism Simulator Engine

The existing implementation is integrated in the Java package *simulator*. The interface methods for approximate model checking and path generation and management, which are used by graphical simulator, are organized in class *simulator.SimulatorEngine*, called directly by other PRISM components.
The engine contains five public methods which are essential in the process of property verification. Their declarations are presented in the Listing 4.1.

These functions allows for crucial operations:

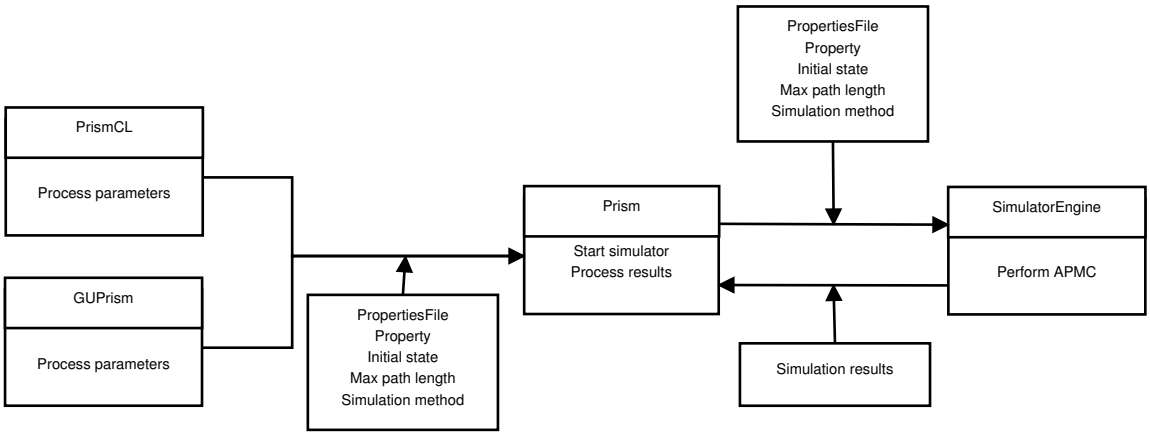
- **Is this model suitable for simulation?**
  If model type can not be handled by the simulator, an exception with error message is thrown.
- **Is this model suitable for simulation?**
  Allows checking if the property is compatible with simulator, resulting with a Boolean result or an exception with an error message.
- **Model checking of one property/multiple properties**
  Approximate model checking of properties.
- **Model checking experiment**
  Model checking for different values for constants.

```
/*
* Model verification.
*/
public void checkModelForSimulation(ModulesFile modulesFile) throws PrismException;
/*
* Property verification.
*/
public boolean isPropertyOKForSimulation(Expression expr);
public void checkPropertyForSimulation(Expression expr) throws PrismException;
/*
* Model checking methods.
*/
public Object modelCheckSingleProperty(ModulesFile modulesFile, PropertiesFile
    propertiesFile, Expression expr, State initialState, int maxPathLength,
    SimulationMethod simMethod) throws PrismException;

public Object[] modelCheckMultipleProperties(ModulesFile modulesFile, PropertiesFile
    propertiesFile, List<Expression> exprs, State initialState, int maxPathLength,
    SimulationMethod simMethod) throws PrismException;

public void modelCheckExperiment(ModulesFile modulesFile, PropertiesFile
    propertiesFile, UndefinedConstants undefinedConstants, ResultsCollection
    resultsCollection, Expression expr, State initialState, int maxPathLength,
    SimulationMethod simMethod) throws PrismException, InterruptedException;
```

Listing 4.1: SimulatorEngine methods for the approximate model checking.

The data flow between PRISM and SimulatorEngine class is depicted on the Figure 4.1.

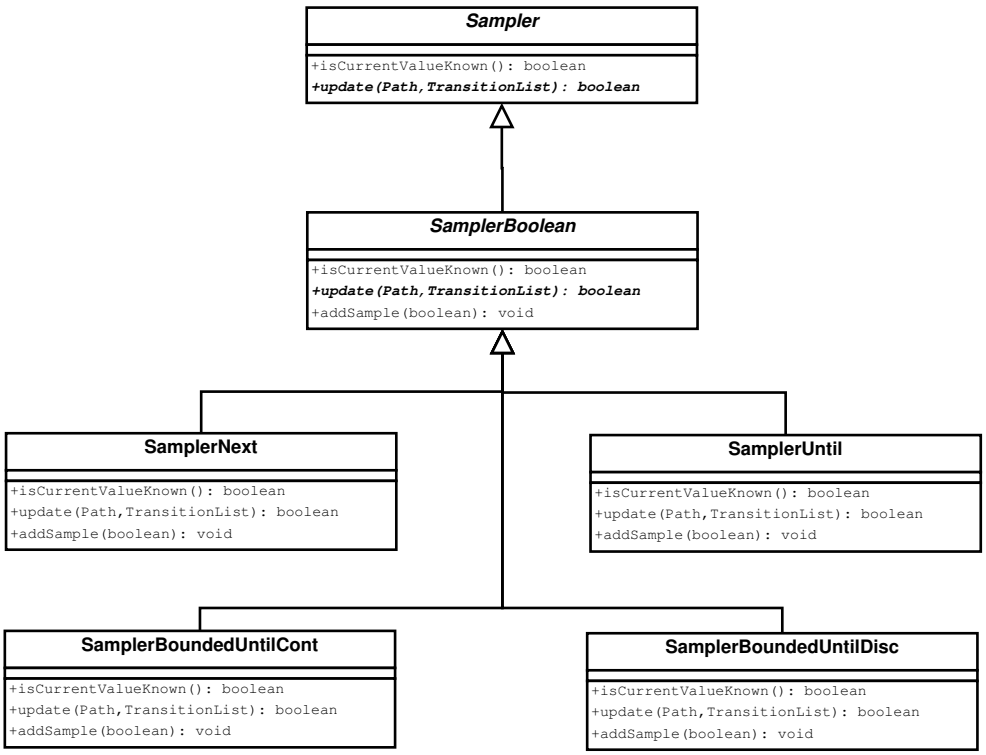
Figure 4.1: Data flow between PRISM and SimulatorEngine.



The process of property evaluation and statistical computations is encapsulated in classes Sampler and SimulationMethod, respectively. The engine creates a Sampler instance, appropriate for provided property. Sampler contains an instance of SimulationMethod which is responsible for different approaches to approximate verification. Figure 4.2 presents the hierarchy of probabilistic samplers with the most important methods; method *update(Path, TransitionList)* is used by SimulatorEngine and the *addSample(boolean)* has been implemented for the purpose of new simulator engine.

In Chapter 3.2, we have provided two algorithms for property approximation: one for the situa-

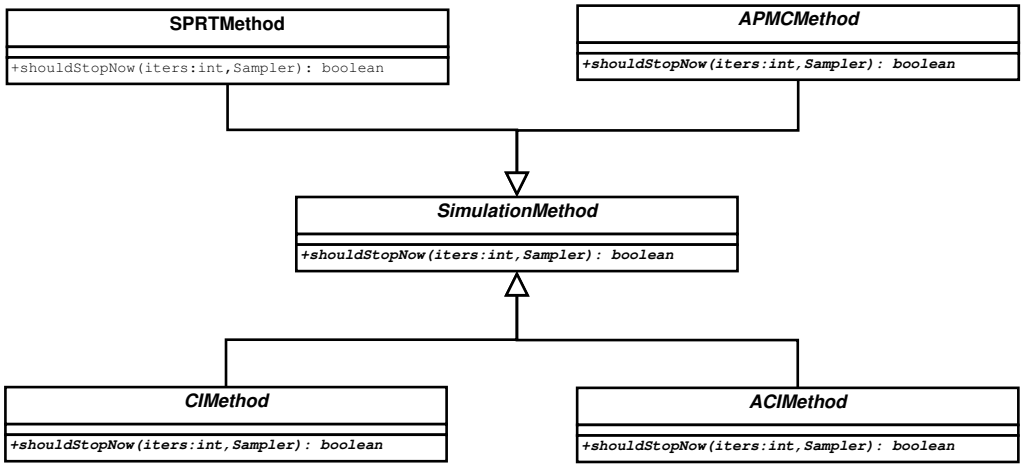
Figure 4.2: Sampler class hierarchy.



tion where sufficient number of samples is known before the simulation and one where it has to be computed during sampling process. The hierarchy of simulation methods in PRISM is presented in Figure 4.3.

The classes *CIMethod* ,*ACIMethod* and *APMCMethod* are abstract and their inheritance tree is very similar: three classes, one for each combination of two specified and one missing parameter. The methods: *CIIterations*, *ACIIterations* and *SPRT* require checking of stop condition after each sample. Other methods work very well with Algorithm 3.

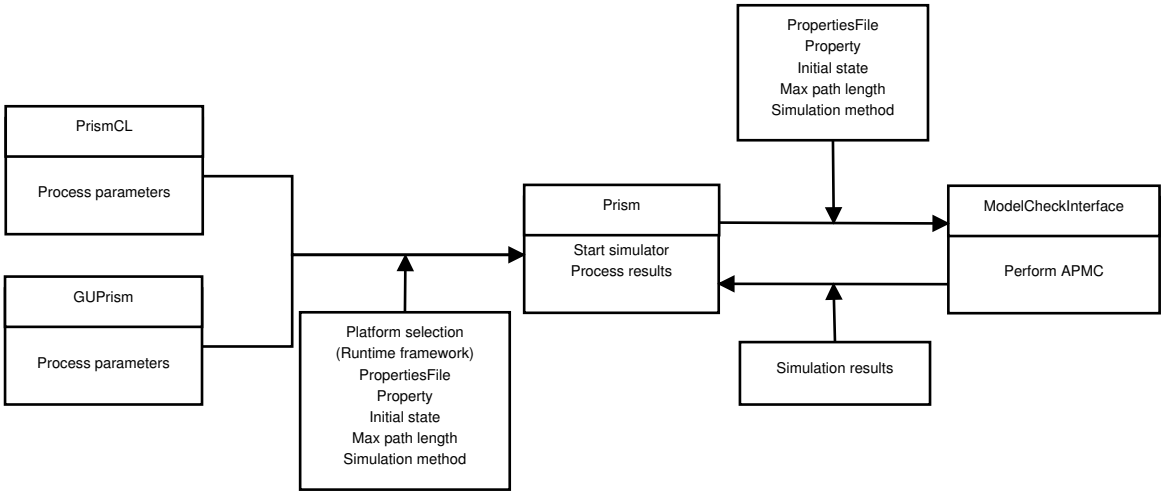
Figure 4.3: SimulationMethod class hierarchy.



4.2 GPU Simulator Engine

To hold the requirement of new simulator being a substitute of the older one, the model checking methods were moved to the new interface *ModelCheckInterface*, implemented by SimulatorEngine and the new *GPUSimulatorEngine*. The data flow between interfaces, PRISM and simulator has changed and the main PRISM class gets information which simulator use; the selection of GPUSimulatorEngine comes with selected framework(described in next paragraph). This small modification in simulation parameters, depicted in Figure 4.4, enables reusage of existing code. New simulators may be easily added in the future, only selection methods in user interfaces has to be extended, and simulator must implement ModelCheckInterface.

Figure 4.4: Data flow with ModelCheckInterface.

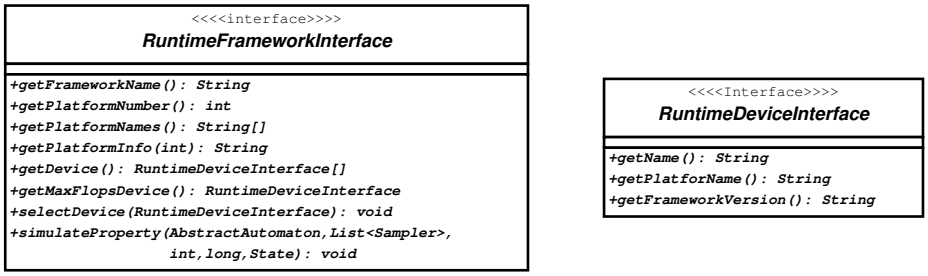


Sampler classes contain algorithms for property verification and new simulator will process this

task in kernel. However, they contain valuable code responsible for extraction property subexpressions, operator and bounds from an Expression object, so it will unreasonable to not use it. SimulationMethod classes will be used directly, without providing any changes.

According to requirements in the previous chapter, new simulator is designed as the union of two parts: model structure and runtime implementations. To enable selection of runtime and devices, two public interfaces were created: *RuntimeFrameworkInterace* and *RuntimeDeviceInterface*, presented in Figure 4.5

Figure 4.5: Runtime and device interfaces in GPU Simulator Engine.



RuntimeFrameworkInterface provides methods for querying platforms and devices, that are available in the system. Device may be selected manually or by using method which will automatically find a device, that will produce the best performance.
RuntimeDeviceInterface contains methods used in device selection in GUI, including device name, platform and the device compute capability i.e. support for the highest version of OpenCL.

4.3 Integration with User Interfaces

Changes in user interfaces were required, to enable easy and simple access to GPU simulator. The main principle of these modifications was to not force users to change their way of working with PRISM. Therefore, by default always existing engine is used. For example, the scripts using the CLI to obtain simulation results will work as in non-modified version of PRISM.

4.3.1 Command-Line Interface

Current simulation options in CLI, shown in Listing 4.2, support choosing simulation method and maximum path length. It was extended to enable simulator selection and basic choice of device, presented in Listing 4.3. By default, the device with maximal number of FLOPS is selected. The device type trigger allows to choose between the best CPU and the best GPU in system. The last supplementary parameter checks if it can find in the system a device with such name; the name is case insensitive and has to be unique. For example, the name 'geforce' is sufficient in case of one CPU and one GPU, but it will cause an error where two NVIDIA's GPU exists in the system.

```
SIMULATION OPTIONS:
-sim ..... Use the PRISM simulator to approximate results of model
    checking
-simmethod <name> ... Specify the method for approximate model checking (ci, aci,
    apmc, sprt)
-simsamples <n> ..... Set the number of samples for the simulator (CI/ACI/APMC
    methods)
-simconf <x> ........ Set the confidence parameter for the simulator (CI/ACI/APMC
    methods)
-simwidth <x> ....... Set the interval width for the simulator (CI/ACI methods)
-simapprox <x> ...... Set the approximation parameter for the simulator (APMC method)
-simmanual .......... Do not use the automated way of deciding whether the variance
    is null or not
-simvar <n> ......... Set the minimum number of samples to know the variance is null
    or not
-simmaxrwd <x> ...... Set the maximum reward -- useful to display the CI/ACI methods
    progress
-simpathlen <n> ..... Set the maximum path length for the simulator
```

Listing 4.2: Simulation parameters in PRISM CLI.

```
-simplatform ........ Select simulation platform. Possible choices: cpu, opencl.
-simdevicetype ...... Specify the device type used by OpenCL simulator. Possible
    choices: cpu, gpu.
-simdevice <name> ... Specify the device used by OpenCL simulator.
```

Listing 4.3: New simulation parameters in PRISM CLI.

4.3.2 Graphical User Interface

The simulation panel, shown in Figure 3.3, has been extended with additional fields for simulator and device selection. The results are shown in Figure 4.6.

4.4 KernelGenerator Library

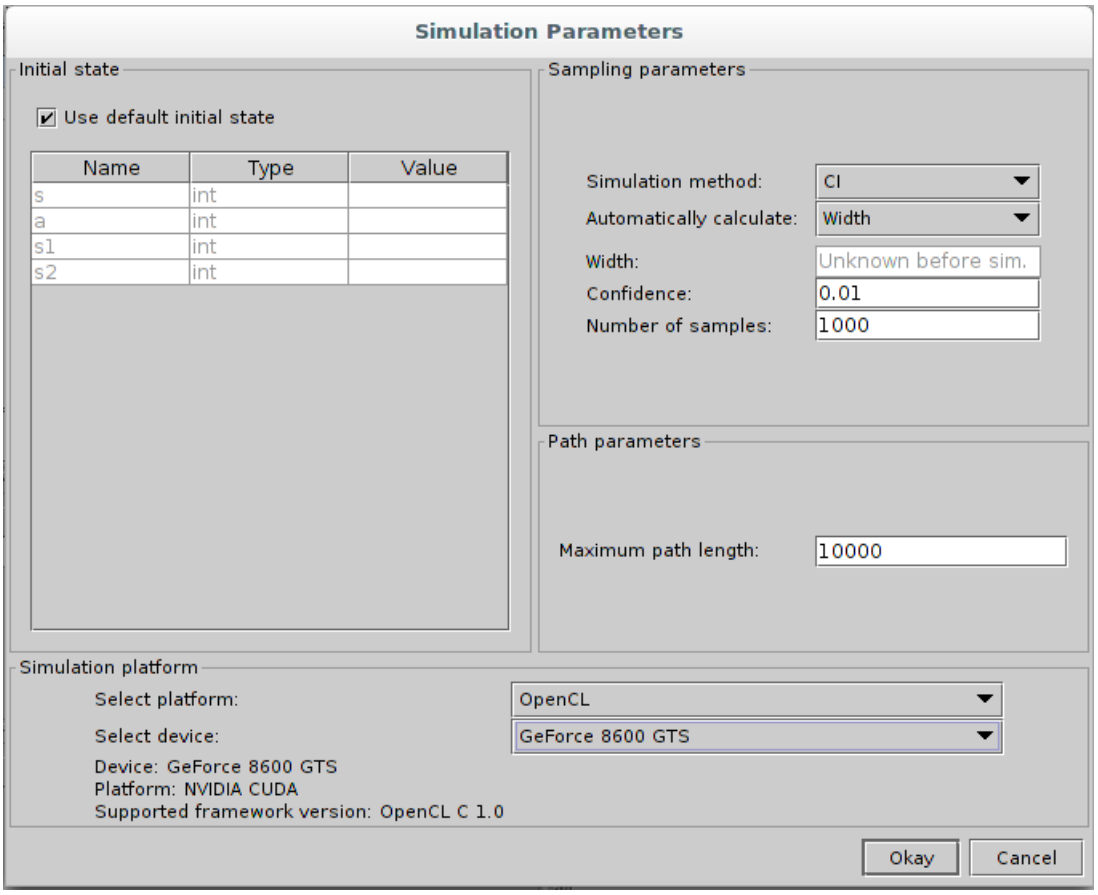
Current simulator engine keeps model structure in memory and generating transitions requires evaluation of each guard. This solution suffers from overhead caused by object-oriented structure of model representation in memory.

Kernel, written in the OpenCL C programming language, has to simulate behaviour of model. The limitations of GPU and OpenCL C does not allow for keeping the model in memory, so the only possible solution is to physically include guards, rates and updates in kernel code. Therefore, for two different models, the kernel will not be the same.

One possible solution is to keep a prototype kernel source code, read it and include model-dependent values into the code. Unfortunately, modifications and improvements, e.g. new model types, will be very hard in this approach. To create more independent way of generating kernel, a library was constructed. KernelGenerator Library consists of two basic parts: *memory*, responsible for built-in and user-defined data types, variables and Rvalues[1], and *expression*, designed for

[1] In C programming language, RValue is the value which can not be the left side of an assignment; e.g. integer literal 42.

Figure 4.6: Modified simulation panel in PRISM GUI.



managing simple expressions and complex components, e.g. loops, branching statements, functions.

Memory

The basic structure of this package is depicted on Figure 4.7. The *VariableInterface* is implemented by classes:

- **StdVariableType**

  Built-in standard variable type. Contains methods for importing PRISM variable into C, with the guarantee of minimal variable size.

- **StructureType**

  Structure in C, containing ordered set of variables.

- **PointerType**

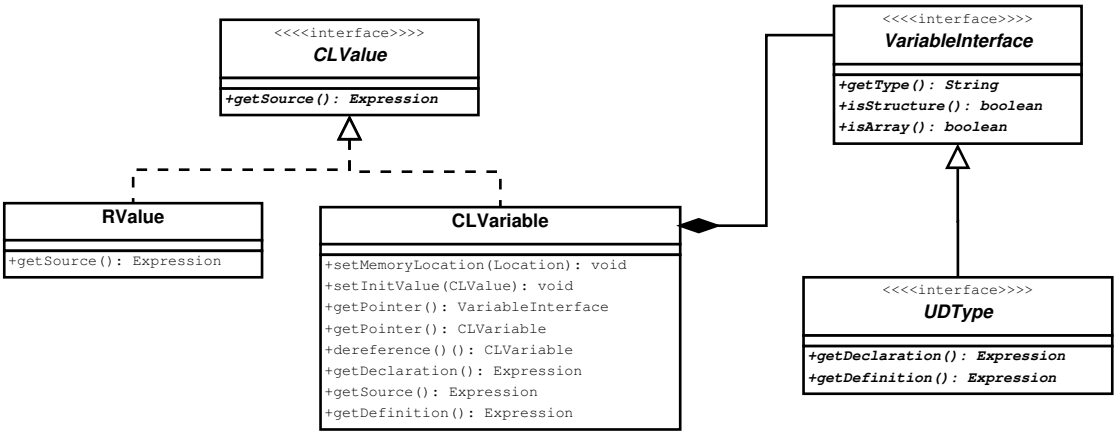
  Pointer to another data type, designed with the *Decorator* design pattern[62].

- **ArrayType**

  Fixed size array, designed with the *Decorator* design pattern.

CLVariable class contains variable name, type, initial value and location in memory - private, local or global. For the purpose of creation complex expressions, e.g. function calls, CLVariable and *RValue* classes implement the common interface *CLValue*.

Figure 4.7: Basic structure of memory classes in the KernelGenerator library.



Expression

The basic structure of this package is depicted on Figure 4.8. The *KernelComponent* is implemented by classes:

- **Expression**

  One, single expression - assignment, function call, logical condition etc.

- **Include**

  Contains preprocessing directive to include a file.

- **IfElse/Switch/ForLoop**

  Represent various constructs from C programming language.

- **Method/KernelMethod**

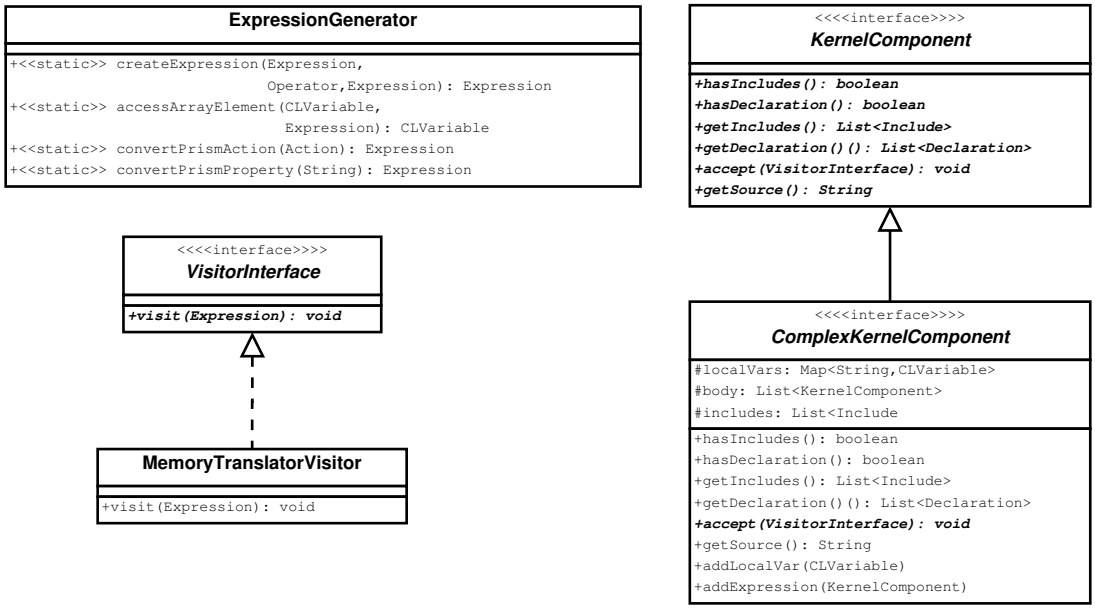
  Creates function with various number of arguments and return statement.

The class *KernelGenerator* consists of many helper methods, used to create arithmetical expressions, accession of structure fields or positions in array, conversion of Prism guards, updates and rates into C.

The *VisitorInterface* provides an interface for classes that 'visit' complex kernel components, according to the *Visitor* design pattern[62]. The *MemoryTranslateVisitor* implements conversion of variables appearing in PRISM expressions, to proper fields in the structure containing state vector of model; this approach was chosen because of different access to state vector in methods.[2]

[2] As it was said in Chapter 2.2.1.2, the state vector structure can not be allocated as a global variable, it has to passed through function arguments.

Figure 4.8: Basic structure of expression classes in the KernelGenerator library.



Chapter 5

OpenCL Implementation

Chapter Abstract

This chapter presents selected details about generated kernels, starting from the schematic view of the main function in the first section. Then the most important tasks are discussed, e.g. synchronized update or loop detection.
Section 5.2 provides simple comparison of tested pseudo-random number generators(PRNGs).

5.1 Kernel design

The generation process results in creating a single source file, containing several structure definitions and function declarations. Only one kernel function is created with arguments:

- **Pseudo-Random Number Generator seed**
  The type and size of this argument strictly depends on used PRNG.
- **Number of simulations**
  Number of samples in current NDRange; the size of NDRange has to be a multiple of local work size, in each dimension, e.g. for local work size equal to 128 and number of samples equal to 10000, the global work size is 10112. This parameter allows to avoid computing additional 112 work-items, which could be dangerous - size of output arrays is equal to the number of samples, and processing overplus work-items may result in writing outside the array.
- **Sample number**
  Number of samples processed in earlier NDRange calls. Used to compute location in output arrays.
- **Path lengths**
  Contains path length of each sample, for computation of minimal, maximal and average path length.

- **Property results**

  Two arrays for each property, one for the result and one to mark if work-item was able to evaluate the property.

Listing 5.1 presents declaration of the kernel function for a case with one property and Random123 as the random number generator.

```
__kernel void main( __global uint* state, uint numberOfSimulations, uint sampleNumber
    , __global uint* pathLengths, __global uchar* property0,__global uchar*
    propertyKnown0);
```

Listing 5.1: The declaration of kernel function.

After variable definitions and PRNG initialization, the main loop begins. Algorithm 5 presents its behaviour. In case of DTMC, the 'count' is just the number of possible transitions; in CTMC is the sum of rates of each transition.

Algorithm 5 Main loop of kernel function.

for $i \leftarrow 0$ **to** $numberOfSamples$ **do**
    generate random numbers, if it is required[1]
    evaluate non-synchronized guards
    evaluate synchronized guards
    $selectionSize \leftarrow$count of independent transitions
    $selectionSynSize \leftarrow$count of synchronized transitions
    if $selectionSize + selectionSynSize == 0$ **then**
        deadlock, end loop
    end if
    update time
    check properties
    if all properties are known **then**
        finished, end loop
    end if
    randomly choose transition and execute it
end for

Next subsection presents a brief description of model representation in OpenCL. Samples of code were generated from the model of a tandem queueing network, shown in Listing 5.2[2].

5.1.1 State Vector Representation

For each model, a structure type is generated, containing all global and local variables. The lower and upper limit for an integer, required in PRISM, is used to compute minimal size of this variable.

[1] Depends on the API of PRNG.
[2] Model taken from PRISM case studies. Source: http://www.prismmodelchecker.org/casestudies/tandem.php

```
ctmc

const int c; // queue capacity

const double lambda = 4*c;
const double mu1a = 0.1*2;
const double mu1b = 0.9*2;
const double mu2 = 2;
const double kappa = 4;

module serverC

  sc : [0..c];
  ph : [1..2];

  [] (sc<c) -> lambda: (sc'=sc+1);
  [route] (sc>0) & (ph=1) -> mu1b: (sc'=sc-1);
  [] (sc>0) & (ph=1) -> mu1a: (ph'=2);
  [route] (sc>0) & (ph=2) -> mu2: (ph'=1) & (sc'=sc-1);

endmodule

module serverM

  sm : [0..c];

  [route] (sm<c) -> 1: (sm'=sm+1);
  [] (sm>0) -> kappa: (sm'=sm-1);

endmodule
```

Listing 5.2: Tandem queueing network model.

```
typedef struct _StateVector {
uchar __STATE_VECTOR_sm;
uchar __STATE_VECTOR_sc;
uchar __STATE_VECTOR_ph;
} StateVector;
```

Listing 5.3: The example of generated state vector.

5.1.2 Transition Representation

For the independent transitions, the indices of active guards are written in an array. This approach allows for immediate selection of update.

Synchronized transitions require more complicated method. For each action label, a structure is generated; containing size of transition in all modules, array with sizes of transition in each module and an array of Boolean flags, to mark active guards. Listing 5.4 presents structure type for action label *step*.

```
typedef struct _SynState__route{
float size;
float moduleSize[2];
bool guards[3];
} SynState__route;
```

Listing 5.4: The structure type for synchronized label.

5.1.3 Synchronized Update

The description presents an interpretation of synchronized update in CTMC; in DTMC equations are very similar, with the difference that probability is in interval $[0, 1)$ and for multiple commands in the module the choice is probabilistic, rather than a race condition. When a synchronized update is selected, a randomized rate r is in interval $[0, size)$. For n pairs of rate and update in the first module, we may interpret this interval as union of n parts, where i-th subinterval is in the form:

$$[\frac{size}{moduleSize[0]} \sum_{0<=k<i} \boldsymbol{R}_k, \frac{size}{moduleSize[0]} \sum_{0<=k<=i} \boldsymbol{R}_k] \tag{5.1}$$

By dividing this value by the size of other modules:

$$\frac{size}{moduleSize[0]} \tag{5.2}$$

we can obtain rate r in bounds proper for the first module. After processing the update, we need to transform the rate into an interval:

$$[0, R_i \frac{size}{moduleSize[0]}] \tag{5.3}$$

Therefore, selection of j-th update gives an equation:

$$r' \ = \ \frac{r - \sum_{0<=k<j} R_k}{R_j} \tag{5.4}$$

This process is repeated for each module in synchronized transition.

Listing 5.5 presents generated functions for synchronized update. The loop in function *updateSyn___route* checks which guard in module *serverC* should be selected; this code is necessary when there is more than one labeled command in module. In the module *serverM* is only one synchronized command, so the function *updateSynchronized___route* can be called directly.

Synchronized update consists of many single steps, one for each module. It is possible that command will use value of variable from previous module and if this variable has been modified in the previous step, then using new value would lead to behaviour that is not compatible with PRISM. For this situation, a copy of the state vector has to be created.

5.1.4 Loop Detection

The states where transition choice is deterministic i.e. there is only one way to take, should be detected during simulation. Sample, which enters this state, will stay in it until maximal path length is reached and the evaluation of unbounded property will not change. For example, average path length in die program, presented in Chapter 3.1.1, is equal to 4 because of six final states - detection of such situations will drastically increase performance, especially when maximal path length is large.

The update methods are responsible for checking whether state vector and its copy are the same.

```
void updateSyn__route( StateVector* sv, SynState__route* synState, float prop) {
    float newSum = 0.0;
    uchar guard = 0;
    float sum = 0.0;
    StateVector oldSV = (*sv);
    float totalSize = (*synState).size;
    totalSize /= (*synState).moduleSize[0];
    prop /= totalSize;
    guard = 0;
    sum = 0.0;
    newSum = 0.0;
    for(;;++guard){
        switch(guard){
            case 0:
                newSum = (1.8) ;
            break;
            case 1:
                newSum = (2.0) ;
            break;
        }
        newSum *= (*synState).guards[guard + 0];
        if(prop < sum + newSum){
            prop -= sum;
            break;
        }
        sum += newSum;
    }
    updateSynchronized__route(sv,&oldSV,(*synState).guards,0,guard,&prop);
    prop *= totalSize;
    totalSize /= (*synState).moduleSize[1];
    prop /= totalSize;
    guard = 0;
    updateSynchronized__route(sv,&oldSV,(*synState).guards,1,guard,&prop);
    prop *= totalSize;
}
void updateSynchronized__route( StateVector* sv, StateVector* oldSV, bool* guards,
    uchar module, uchar guard, float* prob) {
    switch(module){
    case 0:
        switch(guard){
        case 0:
            (*sv).__STATE_VECTOR_sc = ((float)(*oldSV).__STATE_VECTOR_sc)-1;
        break;
        case 1:
            (*sv).__STATE_VECTOR_ph = 1;
            (*sv).__STATE_VECTOR_sc = ((float)(*oldSV).__STATE_VECTOR_sc)-1;
        break;
        }
    break;
    case 1:
        switch(guardSelection){
        case 0:
            (*sv).__STATE_VECTOR_sm = ((float)(*oldSV).__STATE_VECTOR_sm)+1;
        break;
        }
    break;
    }
}
```

Listing 5.5: The example of generated functions for synchronized update with label *route*.

5.2 Pseudo-random Number Generators

During implementation and testing, three pseudo-random number generators were tested. *Random123* was selected as the final choice, because it is the only one PRNG that meets the criteria of performance and correctness. However, the API in KernelGenerator library, allows for easy addition of new PRNG by implementing only one class, which encapsulates details about PRNG initialization, seed and includes.

5.2.1 mwc64x

The MWC64X[63] is a pseudo-random number generator for OpenCL, which allows for uniform generation of 32-bit integers. It uses two 32-bit word for its internal state, requires only one 64-bit integer as global seed for all work-items and provides functions for skipping numbers and splitting streams. The function in Listing 5.6 can be used to provide seed('baseOffset'), increased with subsequent NDRange calls, and number of generated integers in work-item('perStreamOffset') - each thread will be automatically configured.

```
1 void MWC64X_SeedStreams(mwc64x_state_t *s, ulong baseOffset, ulong perStreamOffset);
```

Listing 5.6: Stream skipping in MWC64X.

Unfortunately, the tests showed that skipping numbers is a serious bottleneck, which dramatically decreases performance.

5.2.2 OpenCL PRNG Library

The OpenCL PRNG Library[64] provides implementations of different random generator, including Mersenne Twister. Each work-item obtains its own value for seed, which has to be computed using functions in C language, provided in the library. Because of their complexity, they were not ported into Java, but connected with the Java Native Interface.
The tests revealed that this library does not work very well when work-items use different amounts of random numbers.

5.2.3 Random123

The Random123[65] is a library providing counter-based random number generators(CBRNGs)[66]. The generator is stateless and the process of number generation in deterministic functions, taking two parameters: *counter* and a *key*. For the purpose of sampling, only three values for seed are required. The Listing 5.7 presents initialization and creation of two random integers.

```
//seed
uint state[3];
threefry2x32_ctr_t ctr = {{0,sampleNumber+globalID+state[0]}};
threefry2x32_key_t key = {{state[1],state[2]}};
threefry2x32_ctr_t rand = threefry2x32(ctr, key);
```

Listing 5.7: Random number generation in Random123.

Chapter 6

Testing

Chapter Abstract

This chapter presents results of testing three selected case studies. First section describes testing environment and the section 6.2 consists of three subsections, each for one case study.
The results are summarized in section 6.3.

6.1 Testing Setup

During development, several testing cases were used: simple models, models from PRISM manual. For the purpose of this dissertation, we have selected three case studies from the PRISM benchmark suite[67]: a DTMC model of security protocol, discussed in section 6.2.1, a DTMC model of von Neumann's technique for constructing reliable computations in section 6.2.2 and CTMC model presenting composition of two network queues, in section 6.2.3.

To present correctness of the results, we have compared value computed by PRISM hybrid engine with the interval obtained from simulator:

$$[value - width, value + width]$$

The performance of new simulator engine will be discussed on the GPU and the CPU, in relation to existing PRISM simulator engine.

Table 6.1 presents a detailed description of a computer, which was used for running tests.

[1] Arch Linux.

Position	Description
OS	Linux[1]
Kernel	3.10.29 x86_64
Java	OpenJDK 1.7.0_51
JavaCL	1.0.0-RC3
SimulatorEngine	
CPU	Intel Core2Duo E6550 @ 2.33 GHz
PRISM	4.1.beta2
GPUSimulatorEngine - CPU	
OpenCL Platform	AMD Accelerated Parallel Processing 2.9
OpenCL Device	Intel Core2Duo E6550 @ 2.33 GHz
OpenCL Version	1.2
GPUSimulatorEngine - GPU	
OpenCL Platform	NVIDIA CUDA 5.5
OpenCL Device	GeForce 8600 GTS
OpenCL Version	1.0

Table 6.1: Specification of testing platform.

6.2 Case Studies

Table 6.2 presents simulation parameters, valid for each case study. For the new simulator engine, we will difference *kernel time*, i.e. time of all NDRange calls, and *total time*, which includes also time of kernel compilation and allocating memory objects on device. By default, we use the total time.

Parameter	Value
Simulation method	Confidence Interval
Confidence	0.05
Number of Samples	1 000 000
Width	**?**

Table 6.2: Simulation parameters.

6.2.1 Probabilistic Contract Signing Protocol

One of the potential risking situations in the Internet is exchanging data, when the participants of exchange do not trust each other. The problem of *fair exchange* requires a form of exchange, in which each client obtains his part of the deal, or no one obtains anything. The *contract signing* is a type of protocol, in which both parties agree to *a contract*.

The most important property in contract signing protocol is *fairness*, i.e. protocol is *fair* if it guarantees that obtaining first party's secret by the second participant will always result in first party obtaining second party's commitment.

We will test the contract signing protocol of Even, Goldreich, and Lempel, known as the EGL protocol, verified in [33].

The full model was included in Appendix A, in Listing A.1.

The property computes probability that party A is unfairly disadvantaged, i.e. party B knows A's secrets, but A does not know B's secrets:

$$P_{=?}\ [\ \boldsymbol{F}\ !"knowA" \ \&\ "knowB"\ ] \tag{6.1}$$

We perform computations for different values of constants **K** and **L**, i.e. number of pairs of secrets and number of bits in each secret. Table 6.3 presents results of verification, an absolute error of this value and computed width of an interval, which is the maximal bound for the absolute error. Table 6.4 contains times of approximate model checking for current simulator engine and for

Nr	K,L	PRISM value	Simulation result	Absolute error	Width of confidence interval	Correct?
1	5,2	0.515625	0.514942	0.000683	0.001287	✓
2	10,2	0.50048828125	0.499698	0.00079028125	0.001287	✓
3	15,2	0.5000152587890625	0.499285	0.00073025878	0.001287	✓
4	20,2	0.5000004768371582	0.499269	0.00073147683	0.001287	✓

Table 6.3: Results of approximation - EGL protocol.

the new engine, running on CPU and GPU. All values of time are given in seconds.

Nr	K,L	SimulatorEngine	GPUSimulatorEngine CPU	GPUSimulatorEngine GPU
1	5,2	640.9	8.30	4.68
2	10,2	1294.38	15.11	9.22
3	15,2	1826.34	21.70	14
4	20,2	2489.31	29.08	21.74

Table 6.4: Performance of simulators - EGL protocol.

Table 6.5 shows details about GPUSimulatorEngine: local work size for simulation, time of kernel computation and total time.

Nr	K,L	Local work size	Time of kernel execution	Time
CPU				
1	5,2	1024	6.98	8.30
2	10,2	1024	13.83	15.11
3	15,2	1024	20.4	21.70
4	20,2	1024	27.76	29.08
GPU				
1	5,2	128	4.34	4.68
2	10,2	128	8.84	9.22
3	15,2	128	13.57	14
4	20,2	128	18.69	21.74

Table 6.5: Detailed view of OpenCL simulations - EGL protocol.

6.2.2 NAND Multiplexing

This case study, taken from [68], concerns a technique known as the NAND multiplexing, introduced in 1952 by von Neumann. The method proposes replacing a processing unit with multiplexed units, each containing $\boldsymbol{N}$ copies of each unit. The new unit consists of two stages: the executive stage, responsible for computing N values, which should be equal when devices and input are reliable, and $\boldsymbol{K}$ restorative stages, which tries to reduce errors introduced by faulty units.
The main objective of this method is to create correct and trustworthy results, using unreliable devices. For details, please refer to [68] or to PRISM website. The full model was included in Appendix A, in Listing A.2.

The property computes probability that less than 10 percent of outputs are erroneous, i.e. is the output value reliable?

$$P_{=?}\ [\ \boldsymbol{F}\ s = 4\ \&\ \frac{z}{N}\ <\ 0.1\ ] \tag{6.2}$$

We perform computations for different values of constants **N** and **K**. Table 6.6 presents results of verification, an absolute error of this value and computed width of an interval, which is the maximal bound for the absolute error. Table 6.7 contains times of approximate model checking for current

Nr	N,K	PRISM value	Simulation result	Absolute error	Width of confidence interval	Correct?
1	20,1	0.28641904	0.286554	0.00013496	0.001164	✓
2	20,2	0.41286262	0.413273	0.00041038	0.001268	✓
3	20,3	0.46854396	0.468551	0.00000704	0.001285	✓
4	20,4	0.49415805	0.49431	0.00015195	0.001287	✓
5	40,1	0.28648730	0.28663	0.0001427	0.001164	✓
6	40,2	0.48380547	0.484005	0.00019953	0.001287	✓
7	40,3	0.57770691	0.577464	0.00024291	0.001272	✓
8	40,4	0.61868222	0.61842	0.00026222	0.001251	✓
9	60,1	0.26946099	0.270023	0.00056201	0.001143	✓
10	60,2	0.51753355	0.517503	0.00003055	0.001287	✓

Table 6.6: Results of approximation - NAND multiplexing.

simulator engine and for the new engine, running on CPU and GPU. All values of time are given in seconds.

Tables 6.8 and 6.9 show details about GPUSimulatorEngine: local work size for simulation, time of kernel computation and total time.

6.2.3 Tandem Queueing Network

This case study uses a simple tandem queueing network, consisting of a $\boldsymbol{M/Cox_2/1-queue}$ composed with a $\boldsymbol{M/M/1-queue}$. Both queues have capacity c.
For details, please refer to [69] or to PRISM website. The full model was included in Appendix A, in Listing A.3.

Nr	N,K	SimulatorEngine	GPUSimulatorEngine CPU	GPUSimulatorEngine GPU
1	20,1	588.31	7.26	2.25
2	20,2	989.96	11.55	3.16
3	20,3	1384.01	15.65	3.67
4	20,4	1786.37	19.73	4.37
5	40,1	1201.5	13.52	3.27
6	40,2	1982.39	21.72	4.75
7	40,3	2825.19	29.44	6.21
8	40,4	3526.06	37.73	7.51
9	60,1	1758.56	19.84	4.37
10	60,2	2946.68	31.64	6.34
11	60,3	4119.82	44.27	8.67
12	60,4	5171.35	56.79	10.55

Table 6.7: Performance of simulators - NAND multiplexing.

Nr	N,K	Local work size	Time of kernel execution	Time
1	20,1	1024	6.42	7.26
2	20,2	1024	10.65	11.55
3	20,3	1024	14.84	15.65
4	20,4	1024	18.92	19.73
5	40,1	1024	12.74	13.52
6	40,2	1024	20.91	21.72
7	40,3	1024	28.71	29.44
8	40,4	1024	36.96	37.73
9	60,1	1024	19.12	19.84
10	60,2	1024	30.89	31.64
11	60,3	1024	43.49	44.27
12	60,4	1024	55.91	56.79

Table 6.8: Detailed view of OpenCL simulations on CPU - NAND multiplexing.

For testing we have used below property:

$$P_{=?}\ [\, \boldsymbol{F}\ <=\ 500\ \ sc = c \,\&\, sm = c \,\&\, ph = 2\,] \tag{6.3}$$

The property computes probability that the network becomes full within 500 time units.

We perform computations for different values of constant **c**. Table 6.10 presents results of verification, an absolute error of this value and computed width of an interval, which is the maximal bound for the absolute error.

Table 6.11 contains times of approximate model checking for current simulator engine and for the new engine, running on CPU and GPU. All values of time are given in seconds.

Tables 6.12 and 6.13 show details about GPUSimulatorEngine: local work size for simulation, time of kernel computation and total time.

Nr	N,K	Local work size	Time of kernel execution	Time
1	20,1	448	0.98	2.25
2	20,2	448	1.66	3.16
3	20,3	448	2.35	3.67
4	20,4	448	3.11	4.37
5	40,1	448	2.04	3.27
6	40,2	448	3.4	4.75
7	40,3	448	4.77	6.21
8	40,4	448	6.14	7.51
9	60,1	448	3.02	4.37
10	60,2	448	5.07	6.34
11	60,3	448	7.19	8.67
12	60,4	448	9.24	10.55

Table 6.9: Detailed view of OpenCL simulations on GPU - NAND multiplexing.

Nr	c	PRISM value	Simulation result	Absolute error	Width of confidence interval	Correct?
1	4	0.8695527115894442	0.869393	0.0001597	0.000867	✓
2	5	0.6041677400687193	0.60366	0.0005077	0.001259	✓
3	6	0.3433367462570954	0.342951	0.0003857	0.001222	✓
4	7	0.17362040073689963	0.173181	0.0004394	0.000974	✓
5	8	0.08278767478063223	0.082388	0.0003996	0.000708	✓
6	9	0.038383518654576225	0.038569	0.0001854	0.000496	✓

Table 6.10: Results of approximation - Tandem Queueing Network.

6.3 Analysis

6.3.1 Correctness of results

For each testing case, the results were very close to the actual value. Moreover, the bound given by the confidence interval is often much greater than the real error.

6.3.2 Performance differences

The results revealed that the new simulator performs much better than the existing one. In some cases, the OpenCL implementations are able to verify property within two minutes, and SimulatorEngine needs over two hours for that task.

In terms of CPU and GPU comparison, the GPU scales well when the model is not growing physically i.e. more generated guards, line of codes, etc. For example, the verification time for Tandem Queueing Networks is almost the same for the smallest and the largest case. On the other hand, the computation time for OpenCL CPU is growing more than twice. In the case of EGL, where the increase of size code is significant, the computation time on GPU is growing rapidly.

Nr	c	SimulatorEngine	GPUSimulatorEngine CPU	GPUSimulatorEngine GPU
1	4	3120.76	52.09	19.61
2	5	5238.15	79.45	19.76
3	6	6406.03	100.12	20.27
4	7	7010.82	112.24	20.43
5	8	7053.25	118.96	20.52
6	9	8026.32	121.14	20.62

Table 6.11: Performance of simulators - Tandem Queueing Network.

Nr	c	Local work size	Time of kernel execution	Time
1	4	1024	51.32	52.09
2	5	1024	79.45	79.45
3	6	1024	99.36	100.12
4	7	1024	111.51	112.24
5	8	1024	118.25	118.96
6	9	1024	120.36	121.14

Table 6.12: Detailed view of OpenCL simulations on CPU - Tandem Queueing Network.

Nr	c	Local work size	Time of kernel execution	Time
1	4	320	18.28	19.61
2	5	320	19.35	19.76
3	6	320	19.81	20.27
4	7	320	19.99	20.43
5	8	320	20.09	20.52
6	9	320	20.09	20.62

Table 6.13: Detailed view of OpenCL simulations on GPU - Tandem Queueing Network.

Chapter 7

Conclusions

7.1 Summary of Project

The main objective of this project was to implement parallel, GPU-accelerated simulator engine for the PRISM model checker. The results presented in Chapter 6 show that using new simulator engine, for the approximate verification of PCTL/CSL properties over DTMCs and CTMCs, will give a huge gain of performance. PRISM was enhanced with new structure for simulator engines, containing class hierarchy for manipulating model.

There have been many small problems and bugs that appeared during development. Proper test cases, including small examples of new features, allowed for fast detection of possible errors in the code. Two the most time-consuming issues were:

1. **Bug in library**

   We have lost many hours on tracking down the source of weird and completely unexpected behaviour of kernel. The wrong selection of randomized update was caused by the fact that floating-point literals were incorrectly interpreted and during compilation, the decimal part was removed; e.g the literal 0.99 has been changing into 0. Therefore, some of the update selection conditions were always false, as it is shown in Listing 7.1

```
// selection is a floating-point value in interval [0,1)
if(selection < 0.5) {
    //never selected!
}
else if(selection < 1.0) {
    //always selected!
}
```

Listing 7.1: The example of incorrectly working selection of update.

   The real source of this problem is strange. The system, on which simulator was being developed, has English language, but the polish keyboard. Consequently, the **LANG** was set to 'en_US.UTF-8', but most of the other *locale* parameters were set to 'pl_PL.UTF-8' and

one of them was **LC_NUMERIC**, which is responsible for decimal mark. Contradictory to English language, in Polish the integral and fractional parts are separated with comma rather than with a dot. Due to a bug in JavaCL, this system's property was affecting the source code and that should never happen. As the result, the fractional parts were missing and everything went back to normal after setting **LC_NUMERIC** to 'en_US.UTF-8'.

2. **Selection of PRNG**

   The first chosen PRNG was MWC64X and the kernel implemented with that the generator was suffering from low performance. Many potential sources of this problem were checked, including allocation and copying of memory objects on the device, but it took a long time before it was found out that the bottleneck is the skipping stream function from the PRNG.
   The next PRNG was the Mersenne Twister implementation from OpenCL PRNG library. We have not foreseen the potential problems, so we lost time on connection of this PRNG with PRISM, which includes calling initialization function via the JNI, before realizing that this PRNG works properly only if the number of generated randoms is equal in each work-item.
   The third PRNG, Random123, fulfills the requirements of fast and correct generation of random numbers in OpenCL.

7.2 Deficiencies and Future Work

This section provides main deficiencies of the new simulator engine and possible ideas for future improvements:

- The simulator does not support process algebra, used in PRISM for changing synchronisation of modules. Moreover, this feature is not implemented in the existing simulator engine. Implementing this part of PRISM language in OpenCL kernel is rather nontrivial task and there are not many case studies, that use process algebra[55].
- Currently, rewards are not supported; we did not want to extend this dissertation with the Probabilistic Reward Computation Tree Logic (PRCTL) and Continuous Stochastic Reward Logic (CSRL). This feature is not complicated and it is going to be implemented in the near future.
- The simulator could be enhanced with support for multiple devices, e.g. concurrent sampling on CPU and GPU. It would require creating a load-balancing mechanism.

Appendix A

Case Studies

A.1 Probabilistic Contract Signing Protocol

```
// Randomized Protocol for Signing Contracts [Even, Goldreich and Lempel 1985]
// Gethin Norman and Vitaly Shmatikov 2004

dtmc
const int N; // number of pairs of secrets
const int L; // number of bits in each secret
// B knows a pair of secrets
formula kB = ( (a0=L & a5=L) | (a1=L & a6=L) | (a2=L & a7=L) | (a3=L & a8=L) | (a4=L
    & a9=L));
// A knows a pair of secrets
formula kA = ( (b0=L & b5=L) | (b1=L & b6=L) | (b2=L & b7=L) | (b3=L & b8=L) | (b4=L
    & b9=L));

module protocol
  b : [1..L]; // counter for current bit to be send
  n : [0..N-1]; // counter for which secret to send
  phase : [1..4]; // phase of the protocol
  // 1 - sending secrets using OT (step 1 of the protocol)
  // 2 - send bits of the secrets 0,...,N-1 (step 2 of the protocol)
  // 3 - send bits of the secrets N,...,2N-1 (step 2 of the protocol)
  party : [1..2]; // which party sends next (1 - A and 2 - B)
  // STEP 1
  // A sends
  [receiveB] phase=1 & party=1 -> (party'=2);
   // B sends and we move onto the next secret
  [receiveA] phase=1 & party=2 & n<N-1 -> (party'=1) & (n'=n+1);
  // B sends and finished this step
  [receiveA] phase=1 & party=2 & n=N-1 -> (party'=1) & (phase'=2) & (n'=0);
  // STEP 2 (A sends)
  // next secret
  [receiveB] phase=2..3 & party=1 & n<N-1-> (n'=n+1);
  // move onto secrets N,...,2N-1
  [receiveB] phase=2 & party=1 & n=N-1 -> (phase'=3) & (n'=0);
  // move onto party B
  [receiveB] phase=3 & party=1 & n=N-1 -> (phase'=2) & (party'=2) & (n'=0);
  // STEP 2 (B sends)
  // next secret
  [receiveA] phase=2..3 & party=2 & n<N-1 -> (n'=n+1);
  // move onto secrets N,...,2N-1
  [receiveA] phase=2 & party=2 & n=N-1 -> (phase'=3) & (n'=0);
   // move onto next bit
  [receiveA] phase=3 & party=2 & n=N-1 & b<L
  -> (phase'=2) & (party'=1) & (n'=0) & (b'=b+1);
  // finished protocol
  [receiveA] phase=3 & party=2 & n=N-1 & b=L -> (phase'=4);
  [] phase=4 -> (phase'=4); // FINISHED
```

```
endmodule

module partyA
  // bi the number of bits of B's ith secret A knows
  // (keep pairs of secrets together to give a more structured model)
  b0 : [0..L]; b5 : [0..L];
  b1 : [0..L]; b6 : [0..L];
  b2 : [0..L]; b7 : [0..L];
  b3 : [0..L]; b8 : [0..L];
  b4 : [0..L]; b9 : [0..L];
  // first step (get either secret i or (N-1)+i with equal probability)
  [receiveA] phase=1 & n=0 -> 0.5 : (b0'=L) + 0.5 : (b5'=L);
  [receiveA] phase=1 & n=1 -> 0.5 : (b1'=L) + 0.5 : (b6'=L);
  [receiveA] phase=1 & n=2 -> 0.5 : (b2'=L) + 0.5 : (b7'=L);
  [receiveA] phase=1 & n=3 -> 0.5 : (b3'=L) + 0.5 : (b8'=L);
  [receiveA] phase=1 & n=4 -> 0.5 : (b4'=L) + 0.5 : (b9'=L);
  // second step (secrets 0,...,N-1)
  [receiveA] phase=2 & n=0 -> (b0'=min(b0+1,L));
  [receiveA] phase=2 & n=1 -> (b1'=min(b1+1,L));
  [receiveA] phase=2 & n=2 -> (b2'=min(b2+1,L));
  [receiveA] phase=2 & n=3 -> (b3'=min(b3+1,L));
  [receiveA] phase=2 & n=4 -> (b4'=min(b4+1,L));
  // second step (secrets N,...,2N-1)
  [receiveA] phase=3 & n=0 -> (b5'=min(b5+1,L));
  [receiveA] phase=3 & n=1 -> (b6'=min(b6+1,L));
  [receiveA] phase=3 & n=2 -> (b7'=min(b7+1,L));
  [receiveA] phase=3 & n=3 -> (b8'=min(b8+1,L));
  [receiveA] phase=3 & n=4 -> (b9'=min(b9+1,L));
endmodule

// construct module for party B through renaming
module partyB=partyA[b0=a0,b1=a1,b2=a2,b3=a3, b4=a4,b5=a5,b6=a6,b7=a7,b8=a8,b9=a9,
    receiveA=receiveB]
endmodule
```

Listing A.1: PRISM model for the EGL protocol.
Source: http://www.prismmodelchecker.org/casestudies/contract_egl.php

A.2 NAND Multiplexing

```
// nand multiplex system
// gxn/dxp 20/03/03

// U (correctly) performs a random permutation of the outputs of the previous stage

dtmc

const int N; // number of inputs in each bundle
const int K; // number of restorative stages

const int M = 2*K+1; // total number of multiplexing units

// parameters taken from the following paper
// A system architecture solution for unreliable nanoelectric devices
// J. Han & P. Jonker
// IEEEE trans. on nanotechnology vol 1(4) 2002

const double perr = 0.02; // probability nand works correctly
const double prob1 = 0.9; // probability initial inputs are stimulated

// model whole system as a single module by resuing variables
// to decrease the state space
module multiplex

  u : [1..M]; // number of stages
  c : [0..N]; // counter (number of copies of the nand done)

  s : [0..4]; // local state
  // 0 - initial state
  // 1 - set x inputs
  // 2 - set y inputs
  // 3 - set outputs
  // 4 - done

  z : [0..N]; // number of new outputs equal to 1
  zx : [0..N]; // number of old outputs equal to 1
  zy : [0..N]; // need second copy for y
  // initially 9 since initially probability of stimulated state is 0.9

  x : [0..1]; // value of first input
  y : [0..1]; // value of second input

  [] s=0 & (c<N) -> (s'=1); // do next nand if have not done N yet
  [] s=0 & (c=N) & (u<M) -> (s'=1) & (zx'=z) & (zy'=z) & (z'=0) & (u'=u+1) & (c'=0);
    // move on to next u if not finished
  [] s=0 & (c=N) & (u=M) -> (s'=4) & (zx'=0) & (zy'=0) & (x'=0) & (y'=0); // finished
     (so reset variables not needed to reduce state space)

  // choose x permute selection (have zx stimulated inputs)
  // note only need y to be random
  [] s=1 & u=1  -> prob1 : (x'=1) & (s'=2) + (1-prob1) : (x'=0) & (s'=2); //
    initially random
  [] s=1 & u>1 & zx>0 -> (x'=1) & (s'=2) & (zx'=zx-1);
  [] s=1 & u>1 & zx=0 -> (x'=0) & (s'=2);

  // choose x randomly from selection (have zy stimulated inputs)
  [] s=2 & u=1 -> prob1 : (y'=1) & (s'=3) + (1-prob1) : (y'=0) & (s'=3); // initially
     random
  [] s=2 & u>1 & zy<(N-c) & zy>0  -> zy/(N-c) : (y'=1) & (s'=3) & (zy'=zy-1) + 1-(zy
    /(N-c)) : (y'=0) & (s'=3);
  [] s=2 & u>1 & zy=(N-c) & c<N -> 1 : (y'=1) & (s'=3) & (zy'=zy-1);
  [] s=2 & u>1 & zy=0 -> 1 : (y'=0) & (s'=3);

  // use nand gate
```

```
  [] s=3 & z<N & c<N -> (1-perr) : (z'=z+(1-x*y)) & (s'=0) & (c'=c+1) & (x'=0) & (y
    '=0) // not faulty
61          + perr    : (z'=z+(x*y))    & (s'=0) & (c'=c+1) & (x'=0) & (y'=0); // von
    neumann fault
  // [] s=3 & z<N -> (1-perr) : (z'=z+(1-x*y)) & (s'=0) & (c'=c+1) & (x'=0) & (y'=0)
    // not faulty
63 //          + perr    : (z'=z+(x*y))    & (s'=0) & (c'=c+1) & (x'=0) & (y'=0); //
    von neumann fault

65  [] s=4 -> true;

67 endmodule

69 // rewards: final value of gate
   rewards
71  [] s=0 & (c=N) & (u=M) : z/N;
   endrewards
```

Listing A.2: PRISM model for the NAND multiplexing.
Source: http://www.prismmodelchecker.org/casestudies/nand.php

A.3 Tandem Queueing Network

```
// tandem queueing network [HKMKS99]
// gxn/dxp 25/01/00

ctmc

const int c; // queue capacity

const double lambda = 4*c;
const double mu1a = 0.1*2;
const double mu1b = 0.9*2;
const double mu2 = 2;
const double kappa = 4;

module serverC

  sc : [0..c];
  ph : [1..2];

  [] (sc<c) -> lambda: (sc'=sc+1);
  [route] (sc>0) & (ph=1) -> mu1b: (sc'=sc-1);
  [] (sc>0) & (ph=1) -> mu1a: (ph'=2);
  [route] (sc>0) & (ph=2) -> mu2: (ph'=1) & (sc'=sc-1);

endmodule

module serverM

  sm : [0..c];

  [route] (sm<c) -> 1: (sm'=sm+1);
  [] (sm>0) -> kappa: (sm'=sm-1);

endmodule

// reward - number of customers in network
rewards "customers"
  true : sc + sm;
endrewards
```

Listing A.3: PRISM model for the Tandem Queueing Network.
Source: http://www.prismmodelchecker.org/casestudies/tandem.php

Bibliography

[1] Mukhtar Adamu, Abdulhameed Alkazmi, Abdulmajeed Alsufyani, Bedour Al Shaigy, Dominic Chapman, and Julian Chappell. London ambulance service software failure. *University of Kent*, 2010.

[2] E. W. Dijkstra. Structured programming. In *SOFTWARE ENGINEERING TECHNIQUES. Report on a conference sponsored by the. NATO SCIENCE COMMITTEE. Rome, Italy, 27th to 31st October 1969.*, 1969.

[3] Baier, C and Katoen, JP. Principles of Model Checking. 2008.

[4] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007, 2002.

[5] Research by Cambridge MBAs for tech firm Undo finds software bugs cost the industry $316 billion a year. www.jbs.cam.ac.uk. URL https://www.jbs.cam.ac.uk/media/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/. Accessed: 2014-01-10.

[6] Frederick P Brooks Jr. *The Mythical Man-Month, Anniversary Edition: Essays on Software Engineering*. Pearson Education, 1995.

[7] David Parker. Probabilistic model checking. http://www.prismmodelchecker.org/lectures/pmc/, 2011. Slides from lecture course. Accessed: 2014-01-25.

[8] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[9] GNU General Public License, version 3. http://www.gnu.org/licenses/gpl.html, June 2007. Accessed: 2014-01-25.

[10] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[11] Mark Phillips. CICS/ESA 3.1 experiences. In *Z User Workshop*, pages 179–185. Springer, 1990.

[12] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, mar 1995. Also available as NASA Contractor Report 4673, August 1995, and to be issued as part of the FAA Digital Systems Validation Handbook (the guide for aircraft certification).

[13] Clarke, Edmund M. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

[14] Clarke, Edmund M and Grumberg, Orna and Peled, Doron A. *Model checking*. MIT press, 1999.

[15] Clarke, Edmund M and Emerson, E Allen. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[16] Queille, Jean-Pierre and Sifakis, Joseph. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[17] ACM Turing Award Honors Founders of Automatic Verification Technology. URL http://www.acm.org/press-room/news-releases-2008/turing-award-07/. Accessed: 2014-01-10.

[18] Klaus Schneider. *Verification of reactive systems: formal methods and algorithms*. Springer, 2004.

[19] Steve Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, and Didier Buchs. High-Level Petri Net Model Checking with AlPiNA. *Fundamenta Informaticae*, 113(3-4):229–264, August 2011. ISSN 0169-2968.

[20] Holzmann, G. The SPIN Model Checker: Primer and Reference Manual, 2004.

[21] Willem Visser and Peter C Mehlitz. Model checking programs with java pathfinder. In *SPIN*, volume 3639, 2005.

[22] Artur Rataj, Bożena Woźna, and Andrzej Zbrzezny. A translator of Java programs to TADDs. *Fundamenta Informaticae*, 93(1):305–324, 2009.

[23] Bryant, Randal E. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[24] McMillan, Kenneth L. *Symbolic model checking*. Springer, 1993.

[25] Clarke, Edmund M and Grumberg, Orna and Hiraishi, Hiromi and Jha, Somesh and Long, David E and McMillan, Kenneth L and Ness, Linda A. Verification of the Futurebus+ Cache Coherence Protocol. In *Proceedings of the 11th International Symposium on Computing Hardware Description Languages and their Applications*, volume 93, pages 15–30, 1993.

[26] Joost-Pieter Katoen. *Concepts, algorithms, and tools for model checking.* IMMD, 1999.

[27] Pnueli, Amir. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[28] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools.* Springer Publishing Company, Incorporated, 2010.

[29] Czachórski, Tadeusz and Pekergin, Ferhan. Diffusion approximation as a modelling tool. In *Network performance engineering*, pages 447–476. Springer, 2011.

[30] David J. Tanenbaum, Andrew S. amd Wetherall. *Computer Networks.* Prentice Hall, 2010.

[31] Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In *Formal methods for performance evaluation*, pages 220–270. Springer, 2007.

[32] M. Duflot, M. Kwiatkowska, G. Norman, D. Parker, S. Peyronnet, C. Picaronny, and J. Sproston. *FMICS Handbook on Industrial Critical Systems*, chapter Practical Applications of Probabilistic Model Checking to Communication Protocols, pages 133–150. IEEE Computer Society Press, 2010. To appear.

[33] G. Norman and V. Shmatikov. Analysis of Probabilistic Contract Signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[34] M. Kwiatkowska and C. Thachuk. Probabilistic Model Checking for Biology. In *Software Safety and Security*, NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2014. To appear.

[35] PRISM model checker website. URL http://www.prismmodelchecker.org. Accessed: 2014-01-10.

[36] H.A. Oldenkamp. Probabilistic model checking: a comparison of tools. Master's thesis, University of Twente, the Netherlands, 2007.

[37] Holger Hermanns, Joost-Pieter Katoen, Joachim Meyer-Kayser, and Markus Siegle. A tool for model-checking Markov chains. *International Journal on Software Tools for Technology Transfer*, 4(2):153–172, 2003.

[38] Joost-Pieter Katoen, Ivan S Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N Jansen. The ins and outs of the probabilistic model checker MRMC. *Performance evaluation*, 68(2):90–104, 2011.

[39] Willliam Feller. *An introduction to probability theory and its applications*, volume 1. John Wiley & Sons, 1968.

[40] Christel Baier. On algorithmic verification methods for probabilistic systems. *Universität Mannheim*, 1998.

[41] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.

[42] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time Markov chains. In *Computer Aided Verification*, pages 269–276. Springer, 1996.

[43] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximative symbolic model checking of continuous-time Markov chains. In *CONCUR'99 Concurrency Theory*, pages 146–161. Springer, 1999.

[44] Vincent Nimal. Statistical Approaches for Probabilistic Model Checking. Master's thesis, Oxford University, 2010.

[45] Frédéric Didier, Thomas A Henzinger, Maria Mateescu, and Verena Wolf. Approximation of event probabilities in noisy cellular processes. In *Computational Methods in Systems Biology*, pages 173–188. Springer, 2009.

[46] F Kojima et al. Simulation Algorithms for Continuous Time Markov Chain Models. *Simulation and Modeling Related to Computational Science and Robotics Technology: Proceedings of SiMCRT 2011*, 37:3, 2012.

[47] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate Probabilistic Model Checking. In *Verification, Model Checking, and Abstract Interpretation*, pages 73–84. Springer, 2004.

[48] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL Programming Guide*. Pearson Education, 2011.

[49] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2*. Newnes, 2012.

[50] The OpenCL C Specification. Technical Report 11, Khronos OpenCL Working Group.

[51] NVIDIA Corporation. OpenCL Programming Guide for the CUDA Architecture, Version 4.1. *CUDA SDK*, 2012.

[52] NVIDIA Corporation. OpenCL Best Practices Guide. *CUDA SDK*, 2011.

[53] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide. *http://developer.amd.com/sdks/AMDAPPSDK/documentation*, 2011.

[54] R. Alur and T. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[55] Andrew Hinton. Simulator for the Probabilistic Model Checker PRISM. Master's thesis, University of Birmingham, 2005.

[56] D. Knuth and A. Yao. *Algorithms and Complexity: New Directions and Recent Results*, chapter The complexity of nonuniform random number generation. Academic Press, 1976.

[57] The Approximate Probabilistic Model Checker website. URL http://sylvain.berbiqui.org/apmc. Accessed: 2014-01-10.

[58] Håkan LS Younes. Ymer: A statistical model checker. In *Computer Aided Verification*, pages 429–433. Springer, 2005.

[59] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *QEST*, pages 251–252, 2005.

[60] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22943-5. doi: 10.1007/978-3-642-22944-2_28. URL http://dx.doi.org/10.1007/978-3-642-22944-2_28.

[61] Paul Jennings, Arka P. Ghosh, and Samik Basu. A two-phase approximation for model checking probabilistic unbounded until properties of probabilistic systems. *ACM Trans. Softw. Eng. Methodol.*, 21(3):18:1–18:35, July 2012. ISSN 1049-331X. doi: 10.1145/2211616.2211621. URL http://doi.acm.org/10.1145/2211616.2211621.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[63] MWC64X website. URL http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html. Accessed: 2014-01-10.

[64] OpenCL PRNG Library website. URL http://theorie.physik.uni-wuerzburg.de/~hinrichsen/software/. Accessed: 2014-01-10.

[65] Random123 website. URL http://www.thesalmons.org/john/random123/releases/1.06/docs/. Accessed: 2014-01-10.

[66] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063405. URL http://doi.acm.org/10.1145/2063384.2063405.

[67] M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.

[68] G. Norman, D. Parker, M. Kwiatkowska, and S. Shukla. Evaluating the reliability of NAND multiplexing with PRISM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1629–1637, 2005.

[69] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.