MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

On Chaining Divine and Prism Model Checkers

BACHELOR THESIS

Kristína Zákopčanová

Brno, spring 2015

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Kristína Zákopčanová

Advisor: doc. RNDr. Jiří Barnat, Ph.D.

Acknowledgement

I would like to thank my supervisor, doc. RNDr. Jiří Barnat, Ph.D., for his time, helpful advice and professional experience he kindly shared with me and that goes beyond this work. Great thanks go to members of ParaDiSe laboratory for their willingness to help and the support they have been to me not only during working on this thesis. Last but not least, I thank my family for their love and great support they provided creatively in so many ways (I especially enjoyed the eatable one).

Abstract

This thesis presents a successfully implemented tool chain for verification of multi-threaded C/C++ programs with probabilities expressed using probabilistic choice operator. The verification is achieved by chaining two unique model checkers; namely, DIVINE and PRISM. In this work, we discuss the technical details of extensions necessary to include in DIVINE in order to enable analysis of probabilistic systems. Afterwards, we describe implementation details of the tool chain. And finally, the overall functionality of the tool chain is demonstrated on a set of simple models. These models are also used for an experimental evaluation.

Keywords

DIVINE, PRISM, probabilistic model checking, Markov Decision Process, Linear Temporal Logic, fairness, LTL fairness assumption, fair scheduler

Contents

1 Introduction

It is very common that computer software, computer hardware, and generally any computerised systems exhibit errors. In order to detect these errors, a considerable number of various approaches and techniques have been developed. For example, testing, simulation, and deductive reasoning.

During past decades, the use of computerised systems has been rapidly increasing, affecting various aspects of human life. This involvement went hand in hand with increasing complexity of these systems, and existing verification techniques proved to be insufficient. Naturally, as we demand systems to be correct and reliable, especially when considering systems that are safety critical, mission critical or economically vital, this called for more elaborate verification techniques.

Until the early 1980's, the prevailing paradigm for verification was a manual one of proof-theoretic reasoning using formal axioms and inference rules oriented towards sequential programs. [1] However, the need for concurrent system verification, ideally while avoiding the difficulties of constructing manual proofs, gave birth to model checking [2], a formal verification method.

Model checking is an automated technique that takes two inputs, a model of the system under consideration, and properties describing the system's desired and undesired behaviours, and systematically checks whether these behaviours are present in the model. The systematic check consists of an exhaustive search of all possible system states, which ensures it can be shown that the model truly satisfies a property being verified. In the case that a property does not hold in a model, a model checker provides a counterexample in the form of an execution path leading to the state violating the examined property.

The studied properties of a system are described using temporal logic, which has proved to be highly suitable for system behaviour description, as it describes the ordering of events in time without introducing time explicitly. [3] Moreover, it is extremely intuitive and mathematically precise at the same time. [4]

Typical properties that can be verified using model checking are of qualitative nature. That means that a model checker gives a yes/no answer to the satisfaction of a certain property. However, not necessarily all system behaviours can be described using qualitative properties. There are sys-

tems where probabilistic aspects need to be taken into consideration. Such systems are, for example, systems with real-time conditions, systems interacting with environment that is naturally probabilistic, systems exhibiting probabilistic behaviour due to the use of randomised algorithms or biological probabilistic systems.

The goal of my work is to enable the formal verification of concurrent systems with probabilities modelled in the C or C++ programming language, achieved by chaining existing formal verification tools; namely, DIVINE [5] and PRISM [6] model checkers.

DIVINE is an LTL model checker with the ability to analyse C/C++ programs with probabilities and generate the state space graph of the program in the form of a Markov Decision Process.

PRISM is the most well established symbolic probabilistic model checker, which provides support for the automated analysis of a wide range of quantitative properties for various types of probabilistic models, Markov decision processes in particular.

The rest of this thesis is structured as follows. In Chapter 2, we present the theory behind probabilistic model checking; namely, Markov Decision Process, Linear Temporal Logic, and an approach to modelling concurrent systems. Chapter 3 describes the workflow of the tool chain, some technical aspects of DIVINE and PRISM, and it demonstrates the use of DIVINE probabilistic and non-deterministic choice function `__divine_choice`. In Chapter 4, we explain how to use `DPtoolchain`, and some technical details of the tool chain are discussed. In the last chapter, we introduce several case studies in order to evaluate the realised tool chain.

2 Preliminaries

In this chapter we will focus on the theoretic background of quantitative model checking. Particularly, we will discuss the underlying representation of systems with probabilities, temporal logic used for specifying properties, modelling concurrent systems by means of interleaving, and the issue of fair scheduling.

As we have seen in the introduction, model checking is an automated verification technique that, given a finite-state model $\mathcal{M}$ and a formal property φ, systematically checks whether the model satisfies given property φ, notation $\mathcal{M} \models \varphi$.

The common modelling formalisms for representing systems with probabilities are discrete Markov chains and Markov Decision Processes. In Markov chains, a choice of a successor state is based only on the probability distribution while in Markov Decision Process, non-determinism is also took into consideration. Therefore, an MDP can be considered as a variant of Markov chains that permits both probabilistic and non-deterministic choices. [4] Since we need to allow for non-deterministic choices, for example when representing concurrency, we use Markov Decision Processes (MDP).

2.1 Markov Decision Process

Definition 1. A Markov decision process is a tuple $\mathcal{M} = (S, Act, P, init, AP, L)$ where

1. S is a countable set of state,
2. Act is a set of actions,
3. $P : S \times Act \times S \to [0, 1]$ is a transition probability function such that for all states $s \in S$ and actions $\alpha \in Act$:

$$\sum_{s' \in S} P(s, \alpha, s') \in \{0, 1\},$$

4. $init \in S$ is the initial state,
5. AP is a set of atomic propositions and $L : S \to 2^{AP}$ is a labelling function.

$Act(s)$ denotes the set of actions that are enabled in the state s, i.e. the set of actions $\alpha \in Act$ such that $P(s, \alpha, t) > 0$ for some state $t \in S$. For any state $s \in S$, we require that $Act(s) \neq \emptyset$ and $\forall \alpha \in Act(s). \sum_{s' \in S} P(s, \alpha, s') = 1$. [4]

The semantics of an MDP could be intuitively described as follows. Given s as the current state, a non-deterministic choice is resolved first, followed by the resolution of a probabilistic choice. That means that an action $\alpha \in Act(s)$ is chosen non-deterministically, leading to a state t with probability $P(s, \alpha, t)$. If $P(s, \alpha, t) > 0$, then we call t an α-successor of s. [7]

In the case that there is just one possible action enabled in state s, we call such a state deterministic. Naturally, if all states in an MDP are deterministic, we call such an MDP a Markov chain.

Paths in an MDP describe potential computations, and since they depend on both the non-deterministic and probabilistic choices in model $\mathcal{M}$, paths and path fragments in MDP are defined as alternating sequences of states and actions. [4]

Definition 2. An infinite path fragment in an MDP $\mathcal{M}$ is an infinite sequence $s_0 \alpha_1 s_1 \alpha_2 s_2 \alpha_3 \ldots \in (S \times Act)^\omega$, written as

$$\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots$$

such that $P(s_i, \alpha_{i+1}, s_{i+1}) > 0$ for all $i \geq 0$.
$Paths(s)$ denotes the set of infinite path fragments that start in state s.

The reasoning behind the probabilities of sets of paths of an MDP relies on the resolution of non-determinism which is performed by a scheduler. In any state s, a scheduler chooses one of the enabled actions $\alpha \in Act(s)$. [4]

Definition 3 (Scheduler)**.** Let $\mathcal{M} = (S, Act, P, init, AP, L)$ be an MDP. A scheduler for $\mathcal{M}$ is a function $\mathcal{D} : S^+ \to Act$ assigning an action $\mathcal{D}(\sigma) \in Act(s_n)$ to every finite run $\sigma = s_0, \alpha_1, \ldots, \alpha_n, s_n$.

2.2 Linear Temporal Logic

The formal description of system properties is provided by Linear Temporal Logic (LTL). We will introduce the syntax and semantics of LTL based on the definitions provided in [4] and [8].

Generally, LTL properties are path formulas composed of atomic propositions, boolean connectors and temporal operators. The basic temporal operators are X, F, G and U.

Definition 4 (LTL syntax)**.** LTL formulas over the set of atomic propositions AP are formed according to the following grammar

$$\varphi ::= \ true \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid X\varphi \mid F\varphi \mid G\varphi \mid \varphi_1 \ U \ \varphi_2$$

where $a \in AP$.

The semantics is defined over paths of an MDP $\mathcal{M}$. Let $\pi = s_0\, \alpha_1\, s_1\, \alpha_2 \ldots$ be a path of an MDP, then let π^i denote a suffix $s_i\, \alpha_{i+1}\, s_{i+1}\, \alpha_{i+2} \ldots$ starting at s_i and $\pi(i) = s_i$. The satisfaction relation is defined as follows:

$$\begin{array}{lcl}
\pi \models true & \iff & \text{always} \\
\pi \models a & \iff & a \in L(\pi(0)) \\
\pi \models \neg\varphi & \iff & \pi \not\models \varphi \\
\pi \models \varphi_1 \wedge \varphi_2 & \iff & \pi \models \varphi_1 \wedge \pi \models \varphi_2 \\
\pi \models X\varphi & \iff & \pi^1 \models \varphi \\
\pi \models F\varphi & \iff & \exists k \geq 0 : \pi^k \models \varphi \\
\pi \models G\varphi & \iff & \forall k \geq 0 : \pi^k \models \varphi \\
\pi \models \varphi_1 \ U \ \varphi_2 & \iff & \exists k \geq 0 : \pi^k \models \varphi_2 \wedge \forall (0 \leq j < k) : \pi^j \models \varphi_1
\end{array}$$

Intuitively, path π satisfies property $X\varphi$ if the formula φ holds in the next state of the path. Property $G\varphi$ holds if φ is true in every state on π. $F\varphi$ means that φ holds at some state on the path π. Lastly, property $\varphi_1 U \varphi_2$ is satisfied when φ_1 holds in every state of π until φ_2 is true.

For more details on the semantics of LTL formulas, I kindly refer the reader to [4] or [8].

LTL properties are commonly used to reason about the quality of concurrent systems. The properties that are used for this purpose can be typically divided into two categories: safety and liveness properties. [9]

A safety property can be intuitively described as "nothing bad should happen". [4] Such property is, for example, the mutual exclusion property which states that no two concurrent processes appear in its critical section at the same time. However, the safety property can be trivially satisfied, and that is when the system does nothing. This issue is encompassed by liveness properties.

The second category, liveness properties, states that "something good should eventually happen". [4] When demonstrated on the mutual exclusion property, the liveness property could be described as "each process that has required to enter a critical section will eventually obtain access".

A detailed formal description of safety and liveness properties can be found in [4].

2.3 Probabilistic Model Checking

After having introduced the concept of MDP and LTL properties, we can define probabilistic model checking as follows.

Given a model $\mathcal{M}$ represented by an MDP and an LTL formula φ describing a behaviour of the model, the probabilistic model checking computes the minimal probability Pr_{min} and the maximal probability P_{max} of all paths of the model $\mathcal{M}$ satisfying the property φ with respect to all schedulers $\mathcal{D}$ for $\mathcal{M}$, written $Pr^{\mathcal{D}}_{min}(\mathcal{M} \models \varphi)$ and $Pr^{\mathcal{D}}_{max}(\mathcal{M} \models \varphi)$.

2.4 Modelling Concurrent Systems

When modelling concurrent systems, the usual approach is to represent such a system by the interleaving model in which all of the events in a single execution are arranged in linear order, which is called an interleaving sequence. [8] Events that are executed concurrently are arbitrarily ordered and we need to consider all the possible orders for the verification and analysis of a system. That means that we need to consider all possible interleavings of concurrent events. As a result, the state space of a concurrent program can be extremely large.

In order to reduce the state space of a program, various reduction techniques can be used. In Section 3.3.2 we discuss the way DIVINE reduces the size of a state space of a program.

2.5 Fairness

An important aspect of verification of concurrent systems is fairness. Fairness is concerned with resolving non-determinism in such a way that it is not biased to constantly ignore a possible option. [4]

To illustrate it on an example, let assume a simple system containing mutual exclusion and two processes and let verify a property of starvation freedom. Starvation is a situation when a process is constantly denied access to resources that are necessary for the thread to make progress. The resolution of non-determinism arisen from the interleaving lies on a scheduler, that is the scheduler selects a process to execute next. During verification, a scheduler can choose such path that one process is constantly ignored and its competitor process is always being selected for execution. However, in practice, this is very unlikely to happen. Therefore, we want to consider only such paths that are executed in some "fair" manner and reflect well a realistic behaviour. For this purpose, we introduce fairness assumption that rule out behaviours that are considered to be unrealistic. In our case, a fairness assumption will be expressed as an LTL formula. For formal description of fairness assumption see [4].

In order to establish fairness in computations considered during model checking, we introduce the notion of a fair scheduler.

Definition 5 (Fair Scheduler)**.** Let $\mathcal{M}$ be a Markov decision process and *fair* an LTL fairness assumption. A scheduler $\mathcal{F}$ for $\mathcal{M}$ is *fair* (with respect to *fair*) if for each state s of $\mathcal{M}$:

$$Pr_s^{\mathcal{F}} = \{\pi \in Paths(s) | \pi \models fair\} = 1$$

The fairness assumption *fair* is *realizable* in $\mathcal{M}$ if there exists some *fair* scheduler for $\mathcal{M}$. [4]

In Linear Temporal Logic, fairness assumption can be encoded syntactically into the formula to be verified.

Definition 6. Let $\mathcal{M}$ be a finite MDP, $fair$ an LTL fairness assumption that is realizable for $\mathcal{M}$ and $\mathcal{F}$ a set of fair schedulers. Then, for each LTL formula φ and state s of $\mathcal{M}$

$$Pr_{min}^{\mathcal{F}}(s \models \varphi) = Pr_{min}(s \models fair \rightarrow \varphi)$$

$$Pr_{max}^{\mathcal{F}}(s \models \varphi) = Pr_{max}(s \models fair \wedge \varphi)$$

3 Verification of C/C++ Programs with Probabilities

When verifying a C/C++ program using model checking, in order to analyse all possible behaviours we need to take into consideration the context in which the program runs. It means that the environment that the program is executed within needs to be modelled, including all the possible user-interface actions. Programs which are put into such context are referred to as closed programs. [10]

When modelling the environment, we need to reproduce the freedom of what could happen. Usually, this is represented by a non-deterministic choice. This approach assures that if the model checker states that a certain property is satisfied within a closed program, then that will always be true regardless of what will happen.

However, as we have already seen, non-deterministic choice can be often insufficient for representing some behaviours realistically. For example, when considering an environment with a very low possibility of failure, if we were limited to using only non-deterministic choice then one of the possible sequences of actions would be an infinite sequence of failures, whereas the real probability of this happening is infinitesimal.

Therefore, we introduce the probabilistic choice as a natural extension of the non-deterministic choice.

In the next section, we will outline the workflow of the realised tool chain, followed by a section which demonstrates both the functionality of the tool chain and use of the non-deterministic and probabilistic choice in C/C++ programs using the DIVINE built-in function `__divine_choice`. In the final section, we will discuss the various technical aspects that play an important role in the quantitative verification tool chain.

3.1 Workflow

The workflow of the tool chain for quantitative verification of C/C++ programs can be divided into three parts. The first part consists of processing C/C++ programs by DIVINE in order to generate an explicit representation of the state space of a program. The explicit representation is used in the second part as an input of the `DPtoolchain` program which transforms it

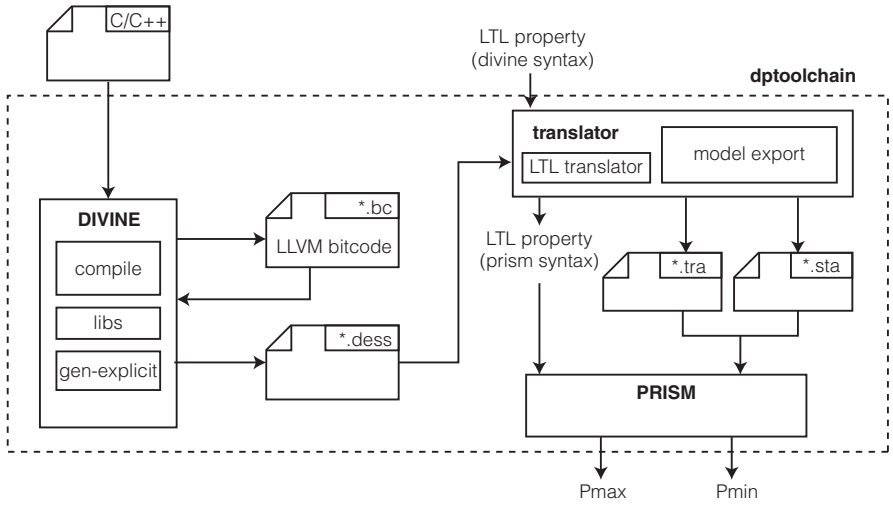

Figure 3.1: Divine – Prism tool chain workflow

into a format acceptable by PRISM, together with an LTL property to be verified. In the last part, we employ the PRISM model checker for quantitative verification of the explicit model against provided LTL formulas. Details of the tool chain are depicted in Figure 3.1

Since DIVINE does not directly understand C/C++ programs, it needs to transform such programs into an input format it can analyse. For this purpose, DIVINE uses LLVM framework; namely, the LLVM intermediate representation format. Thus, a C/C++ program is first translated into an LLVM bitcode (`*.bc`) which is interpreted using a custom interpreter of DIVINE in order to generate the state space graph of the program. The state space representation is saved in a custom file format of DIVINE; namely, DIVINE Explicit State Space (DESS), which is described in detail in Section 3.3.1.

Since the DESS format for representing explicit models is naturally distinct from the representation of explicit models in PRISM, we have created a special program `DPtoolchain` which takes care of the transformation of a model from the DESS representation into the one of PRISM. The PRISM format for representing explicit models is described in Section 3.3.4. `DPtoolchain` is also used to translate LTL properties from DIVINE syntax into the syntax of PRISM.

```
#include<atomic>
std::atomic<int> i;

int main(){
    i=0;
    while (i<4 && __divine_choice(3,3,1,2)) i++;
    assert (i>1);
    return 0;
}
```

Figure 3.2: Example of C program with the probabilistic choice

Now that we have both a model and an LTL property in the format defined by PRISM, we can employ PRISMfor quantitative model checking.

3.2 Introducing Probability in Source Code

Now that we know how the probabilistic verification is enabled in our tool chain, we will demonstrate how to implement probabilistic and nondeterministic behaviour in C/C++ programs.

Let us first assume a simple probabilistic program, shown in Figure 3.2. The program contains a single integer variable `i` whose initial value is `0`. Value of `i` is incremented inside a `while` loop until the variable reaches a value of `3`. However, the while loop can be early-terminated, that is before variable `i` reaches value `3`, with 50% probability in each iteration. This is achieved by using the DIVINE built-in function `__divine_choice` which simulates the probabilistic choice.

The `__divine_choice` function takes a variable number of parameters and returns an integer from the range $\langle 0, n-1 \rangle$, where n is the first and obligatory parameter of the function and states the total number of possible choices, that is the number of state successors created in MDP. Given n probabilistic choices, the `__divine_choice` function needs to receive other n parameters, each setting the probability weight of a respective branch. Therefore, a `__divine_choice` function with n choices receives $n+1$ parameters in total, where $(i+1)$th parameter sets the weight of the ith branch, numbering both branches and parameters from zero. The returned integer value identifies a branch in which the program continues.

The probability of each branch is obtained as its weight divided by the sum of weights of all branches. Thus, the call `__divine_choice(3,3,1,2)` returns 0, 1 or 2 with probabilities $3/6$, $1/6$ and $2/6$. Now, we can clearly see that the `while` loop terminates when `__divine_choice` returns 0, that is with 0.5 probability.

Note that while the program can see just the returned integer value of `__divine_choice`, DIVINE explores all the possible outcomes, which means all three branches created, each entered with the aforementioned probability.

After the termination of the while loop, we test in an assertion statement whether the value of `i` is greater than one. We can see that for the value of `i` to be greater than one, the while loop cannot be early-terminated during the first two iterations.

Having explained the employment of probabilistic choice inside source code, we can use our tool chain for computing the probability of assertion failure. We use an atomic proposition `assert` which indicates whether an assertion is violated in a program. Since there is only one assertion in our program, we can simply ask about the presence of the atomic proposition `assert` in the program at anytime, thus the LTL property will be $F(assert)$. The obtained minimal and maximal probability of assertion violation is 0.75. This can be easily proven to be correct as the probability of the early-termination is 0.5 in the first iteration and 0.5×0.5 in the second iteration, hence 0.75 in total.

We can see that the result of both minimal and maximal probability is the same. This is due to the absence of non-deterministic behaviour in the program. In order to demonstrate the use of both probabilistic and non-deterministic choice in a program, we will slightly modify our program in Figure 3.2 by adding a non-deterministic choice.

The non-deterministic choice is implemented by the `__divine_choice` function as well. However, in contrast to the probabilistic choice, it takes only one parameter that states the number of non-deterministic choices, that means the number of created branches, and returns an integer indicating the resulting branch.

Now, let's assume a program (shown in Figure 3.3) which is derived from the program depicted in Figure 3.2. We can see that the first part of the program has not changed, in each `while` loop the value of `i` is incremented while there is a 50% chance of early-termination. However, a non-deterministic choice `__divine_choice(2)` is entered after the `while`

```
std::atomic<int> i;

int main(){
    i=0;
    while (i<4 && __divine_choice(2,1,1)) i++;
    if (__divine_choice(2))
        assert (i>1);
    else
        assert (i>2);
    return 0;
}
```

Figure 3.3: Example of C program with both probabilistic and non-deterministic choices, `toy0.c`

loop, creating two branches, each with a different assertion statement. The first assertion, `assert(i>1)`, is violated with 0.75 probability, as we saw in the previous example. The second assertion, `assert(i>2)`, is violated with probability 0.875. It is calculated as the sum of the probability of early-termination during first two iterations, that is 0.75, and the probability of early-termination in the third iteration, that is $0.5 \times 0.5 \times 0.5$. When employing the tool chain in order to compute the probability of assertion violation, it runs the verification with respect to the resolution of non-determinism, therefore, the minimal probability of assert violation is 0.75 and the maximal probability is 0.875, as expected.

So far, we have seen the use of both probabilistic and non-deterministic choices on a simple-threaded program. In our last example, we will present a simple multi-threaded program as an extension of the program in Figure 3.2.

Let's take the program in Figure 3.2 and modify it in the following way: before entering the while loop we create a new thread which asynchronously increments a value of `i`, and then we wait for it to terminate by calling `pthread_join` before the assertion statement is reached. We know that in practice, the program will always terminate, and therefore the assertion statement will always be evaluated. However, when such a program is analysed by a model checker, the scheduler can choose such a path that the join operation may never succeed, and thus the program will neither

```
std::atomic<int> i;

void *T1(void *){
    i++;
    return 0;
}

int main(){
    i=0;  pthread_t p;  pthread_create(&p,0,T1,0);
    while (i<4 && __divine_choice(2,1,1))
      i++;
    pthread_join(p,0); AP( joined );
    assert (i>1);
    return 0;
}
```

Figure 3.4: Example of multi-threaded C program with the probabilistic choice, `toy1.c`

terminate nor evaluate the assertion statement. In order to rule out such unrealistic behaviours, model checkers usually include the option to consider only those paths that can be scheduled only by fair schedulers. Such paths are referred to as fair paths. In order to identify fair paths, we introduce a fairness condition that needs to be satisfied by a path to be considered a fair path. We extend our tool to process fairness conditions expressed as an LTL formula. So, in our example of a program in Figure 3.4, the fairness condition will be specified as $F(joined)$. As an atomic proposition `joined` is present in the program after the thread that was created from the main procedure terminated, the satisfaction of the fairness condition makes certain that the whole program will terminate as well.

In our tool chain, there is also an alternative for ensuring fairness verification using verification of only realistic executions using PRISM's implementation of fairness. PRISM enables fairness verification by using the `-fair` switch. As we supply PRISM with only explicit model representation, PRISM cannot have any notion about processes in the original program. Therefore, fair analysis cannot depend on the processes being scheduled. Instead, a path of the MDP is considered to be fair if, for states s occurring infinitely often in the path, each possible non-deterministic choice available in state s is taken infinitely often.

3.3 Technical Details

In this part we discuss the technical details of DIVINE, including some modifications in DIVINE that were necessary to allow probabilistic analysis of C/C++ programs and we describe PRISM's format for explicit models.

3.3.1 DIVINE Explicit State Space (DESS)

DIVINE uses its custom format DIVINE Explicit State Space (DESS) for the representation of a state space of a program. For the purposes of probabilistic model checking, the original DESS format was extended of transition probabilities and state labels. In other words, it was extended so it could encode Markov Decision Processes.

The structure of the DESS format is as follows. Generally, the DESS is composed of two parts, a fixed-length header and a graph representation.

The header contains two types of information. First, it stores general information about the DESS file, such as the DESS version, DESS id string or byte order check which is used to verify the endianity of the DESS file. It also contains model specific information, such as the number of graph vertices, forward edges offset or nodes offset. The exact description of the header format can be found in the DIVINE manual [11].

The second part of the DESS format contains an encoding of the graph which represents an input model.

Vertices of the graph correspond to individual states of an input model, and they are sequentially numbered starting from 1. Moreover, DIVINE adds an extra vertex into a graph, indexed 0, in order to uniquely identify all initial states of the model. Thus, there is an outgoing edge from the vertex indexed 0 to each of the vertices representing the initial states of a model.

Each vertex stores a pointer to a data block. A data block is composed of tuples representing edges. Each tuple represents one outgoing edge from the original vertex into a vertex whose index is stored in the tuple. Originally, a tuple comprised an index of the vertex to which the edge leads and an optional label. However, for the purpose of representing an MDP, the tuple was extended to 3-tuple with the transition probability saved as the third tuple element.

As for the construction of a model from an explicit graph representation, PRISM needs to obtain a definition of each state in terms of a set of atomic propositions, the definition of a state in DIVINE was extended of state flags (state labels). State flags represent respective atomic propositions defined in the program and also some general safety properties automatically verified by DIVINE, such as assertion violation, division by zero, memory leak, mutex deadlock and others. The value of a flag of a specific state is set to `1` if the flag represents an atomic proposition that is present in the state, or if it represents a property that is violated in the state. Otherwise, the state flag is set to `0`. Currently, the number of state flags is limited by the size 64 bit, therefore, each state can only be defined in terms of 64 flags maximum.

3.3.2 Reducing State Space in DIVINE

As we have seen in Section 2.4, the basic approach to modelling concurrent programs is by means of interleaving. In practice, this means the non-deterministic choice of the order in which the actions of the processes that run in parallel are executed. As such non-deterministic choice (i.e. context switch) is basically allowed after each instruction step in any of the threads, and especially when in combination with the very fine-grained LLVM bitcode, the state space of a program grows rapidly. Thus, DIVINE faces the typical problem of state space explosion which can render the whole verification process infeasible.

However, DIVINE takes some measures to overcome the problem of too large state spaces. First of all, it implements several very efficient state-space reduction techniques, such as Tau and Tau+ reductions [12] which are combination of partial order reduction and path compression, which, combined with parallel and distributed-memory processing, renders DIVINE suitable for the verification of large systems. [13]

Moreover, we observe that interleaving at some points does not produce any new results with respect to verified property, and therefore seems pointless.

DIVINE offers the user an option to avoid such pointless context switches by providing him with a tool for creating atomic sections of code. Such sections are delimited using two DIVINE's built-in functions, namely `__divine_interrupt_mask` and `__divine_interrupt_unmask`, the former marking the beginning and the latter marking the end of an atomic section.

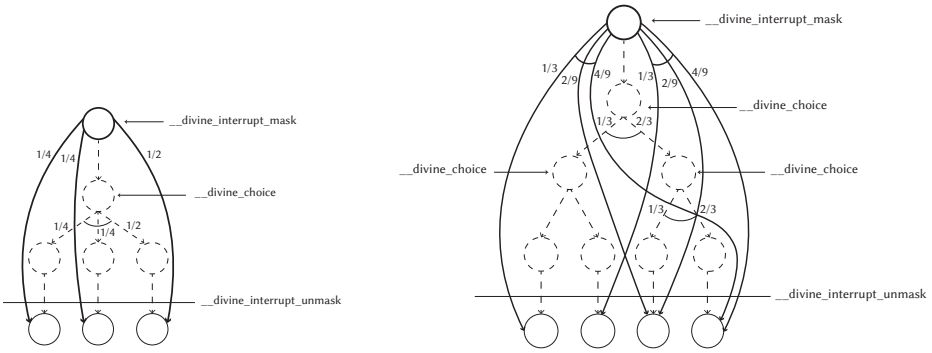

Figure 3.5: a) Branching in an atomic section. b) Nested branching in an atomic section.

In between these two functions, the running thread cannot be interrupted by any other thread, i.e. no context switches are allowed.

In the sequel, the number of considered interleavings in the model of a concurrent system can be significantly decreased. As a result, this reduces the number of states of the corresponding explicit graph representation as no intermediate states need to be created, and therefore, the whole verification process may be rendered faster.

3.3.3 Property Specification in DIVINE

DIVINE supports the verification of properties expressed by LTL formulas, and it also provides the verification of some built-in safety properties.

By default, in every program, DIVINE verifies so called Safety property, which is a combination of assertion, memory, mutex deadlock and arithmetic safeties, memory leak freedom, custom defined safety problems and the safety of compiler-defined guards.

LTL properties to be verified can be specified in two ways. Properties can be either provided to DIVINE, and by extension to `DPtoolchain`, through a command-line argument, or the properties can be defined in the source code of a program and pass to command-line only the name of a property to be verified. The DIVINE syntax for specifying LTL properties follows.

Definition 7 (LTL Syntax)**.** Given a set of atomic propositions AP and $a \in AP$, the syntax of LTL formula φ acceptable by DIVINE is as follows:

```
φ ::= un_op φ |φ bin_op φ | (φ) | term

un_op ::= ! | X | F | G
bin_op ::= && | || | -> | <-> | ^ | U | V | W
term ::= true | false | a
```

The name of atomic proposition is a string of the following structure.

```
{_a-z}[_0-9a-z]*.
```

However, there are some words that are reserved by DIVINE and cannot therefore be used as the names of user-defined atomic propositions. The list of currently reserved words is:
`init`, `noproblem`, `other`, `assert`, `invaliddereference`, `invalidargument`, `outofbounds`, `divisionbyzero`, `unreachableexecuted`, `memoryleak`, `notimplemented`, `uninitialised`, `pointstoviolated`, `deadlock`, `trace`

Atomic propositions need to be defined inside the source code as an enumerated type with the type name `APs`, for example:

```
enum APs { found, lucky, black, white };
```

Their presence in the program is then set by calling a function `AP` which takes the name of an atomic proposition as its only parameter, for example `AP( lucky )`.

3.3.4 PRISM Explicit Model Files

Despite the fact that PRISM is primarily a symbolic model checker focused on verification of models described using PRISM language, PRISM also provides support for verification of imported explicit models. It defines several plain text file formats [14] for explicit model representation. For the purpose of representing an MDP, we will use a state file (`*.sta`) and a transitions file (`*.tra`). PRISM also provides a file format for saving labels (*.lab) with description of states that satisfy them. However, for the given moment, it is

not yet fully implemented. Therefore, we use the state file for saving the state labels as well as for identifying an initial state what could otherwise be saved in the label file. Below, we will describe formats of both state and transitions files.

Transitions file

The transitions file (`*.tra`) contains an explicit list of transitions of a model.

n	c	m	
i_0	k_0	j_0	x_0
i_1	k_1	j_1	x_1
i_2	k_2	j_2	x_2
⋮	⋮	⋮	⋮
i_m	k_m	j_m	x_m

Figure 3.6: PRISM transitions file format

The first line of the file provides information about the model generally; namely, the number of states (`n`), the number of non-deterministic choices (`c`) in the whole model and the total number of transitions (`m`). The remaining lines take the form "`i k j x`", where `i` and `j` are the source and destination state indices of the transition, `k` is the index of the non-deterministic choice that it belongs to, and `x` is the probability of transition. State indices are zero-indexed. [14]

State file

The state file (`*.sta`) contains an explicit list of a model's states.

The first line is of the form $(v_0, v_1, \ldots, v_n)$, containing the names of all atomic propositions that appear in the model. Each of the following lines corresponds to one state whose index is represented by the first number on each line followed by the values of atomic propositions at that state, value x_i corresponding to the v_ith atomic proposition. [14]

Usually, the initial state of a model would be defined in the label file, but since loading initial states from the label file is not yet implemented, we have built a workaround. In the case that PRISM receives no information

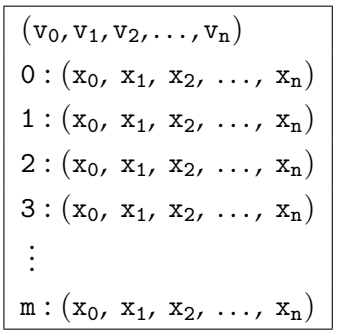

Figure 3.7: PRISM state file format

about the initial state, it automatically considers a state in which all the variables take their minimum value to be the single initial state of a model.

As we saw in Section 3.3.1, DIVINE adds an extra vertex to each graph and considers this vertex to be the initial state of an MDP. Naturally, none of DIVINE's automatically checked properties is violated in this state, nor can an atomic proposition be present in it. Therefore, we know that all the variables take their minimum. However, we have no guarantee that this is an unique state with minimal values of variables in the whole graph. A simple solution to this is to add an extra variable which takes value `0` only in the initial state, otherwise it is evaluated to `1`. At this moment, we know each graph to have a single initial state that can be recognized by PRISM. Hence, in the case of our tool chain, the first variable's name is always `init` and it serves to indicate the initial state of a model. Thus, `init` is a reserved word and cannot be used as the name of any atomic proposition.

3.3.5 PRISM Property Specification Language

Given a set of atomic propositions AP and $\mathtt{a} \in AP$, the syntax of LTL formula φ acceptable by PRISM is as follows:

φ ::= un_op (φ) | φ bin_op (φ) | (φ) | term

un_op ::= ! | X | F | G

bin_op ::= & | | | => | <=> | U | V | W

term ::= true | false | "a"

Then, the syntax of a query about the minimal and maximal probability of a model to satisfy a behaviour described by an LTL property φ is as follows:

```
Pmin = ? [ φ ]
Pmax = ? [ φ ]
```

4 DPtoolchain

The realization of the chaining of DIVINE and PRISM model checkers is implemented by a bash script `DPtoolchain` and `modeltranslator` program written in C++ programming language.

`DPtoolchain` ensures the overall probabilistic model checking functionality by chaining DIVINE, PRISM and `modeltranslator` programs and processing their inputs and outputs.

The `modeltranslator` program serves two purposes. Firstly, it translates the LTL formula from DIVINE syntax into that of PRISM. Secondly, it creates the necessary files which represent a model under verification in a format acceptable by the PRISM model checker.

In the first part of this chapter, we will outline the file organisation of `DPtoolchain`. Afterwards, we will the discuss technical details of its flow, and at the end we will describe how to use it.

4.1 File System Structure

During the execution of the tool chain, there is a non-trivial number of generated files. These files are either necessary for the subsequent steps of the tool chain and verification process or they can reduce the time for later verification of the same model, for instance, when verifying different properties of the same model. Some processes, such as generating the explicit state-space of a program, may take a non-trivial amount of time, so having an option to reduce this time and, in consequence, reduce the time of the verification process, is a highly desired property of the tool chain. In order to do so, it is inevitable to build a logical structure among the files so that both `DPtoolchain` and the user can easily find and work with necessary files.

For every model, `DPtoolchain` creates a special directory whose name is derived from the model's basename. All files produced during the execution of `DPtoolchain` over a specific model are saved in the model's directory. All such model directories are located in the folder `path_to_DPtoolchain_dir/models/` as depicted in Figure 4.1.

All model representations such as the source code, DESS format, state and transitions files bear an identical basename.

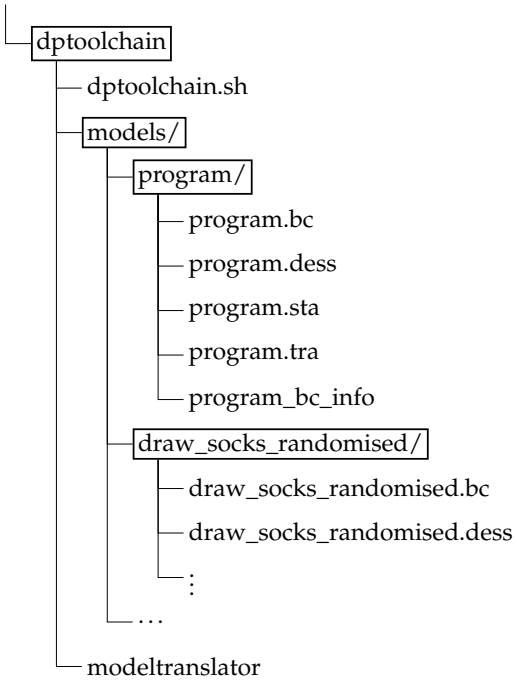

Figure 4.1: `DPtoolchain` file hierarchy

4.2 Using DPtoolchain

When running `DPtoolchain` script, it is necessary to have set paths of DIVINE and PRISM executable files to the PATH variable.

The use of `DPtoolchain` is showed below. Meaning of individual options is described in Table 4.1

```
dptoolchain -i <filename>

dptoolchain -c <filename>  [-f <flags>]
                           [-l <ltl_formula> | -p <property_name>
                           [-s <fair_cond>] ]

dptoolchain -n <filename>  (-l <ltl_formula> | -p <property_name>
                           [-s <fair_condition>] )

dptoolchain -h
```

`-i <filename>`	Print information about a model which is represented either as an LLVM bitcode (`*.bc`) or by DIVINE Explicit State Space file (`*.dess`) Both file formats provide a list of state flags defined for each state. When the input file is an LLVM bitcode, a list of properties defined by DIVINE (general safety properties) plus properties defined in the source code of a program are displayed.
`-c <filename>`	Compile C/C++ program into an LLVM bitcode and generate an explicit state space. In case LTL property is specified, run probabilistic model checking.
`-n <basename>`	Run probabilistic model checking over a compiled model with given basename, it means its state space has been generated. The necessary model files are automatically accessed in the corresponding model directory, therefore it is not necessary to provide path to a specific file.
`-h`	Print help.
`-f <flags>`	Flags that are necessary for compilation of corresponding C/C++ program
`-l <ltl_formula>`	LTL formula of property to be verified.
`-p <property_name>`	The name of LTL property which is specified in a source code of a program under verification.
`-s <fair_condition>`	Specify LTL fairness condition and run probabilistic model checking with respect to it.

Table 4.1: `DPtoolchain` options

Names of atomic propositions representing DIVINE-difined properties are derived from state flag names stated in model information in the following way. A name of an atomic proposition is the string following a colon in a name of a state flag converted to lower case. For example, let the state flag name be `G:Assert` and `G:InvalidArgument`, then valid atomic propositions are `assert` and `invalidargument`.

4.3 Technical Details of the Workflow of DPtoolchain

Compilation

As we have seen in Section 3.1, DIVINE cannot verify C and C++ programs directly. Therefore, the first necessary step is to translate C and C++ code into LLVM bitcode using a suitable C or C++ compiler and then link it against the DIVINE-provided runtime libraries. [11]

However, the compilation of the DIVINE runtime environment takes a non-trivial amount of time. But, as the DIVINE-runtime environment can change only with a new version of DIVINE, we can reduce the time spent on the compilation by pre-building it. Therefore, with each new version of DIVINE, `DPtoolchain` automatically pre-builds DIVINE runtime libraries and then when compiling C/C++ code, it just adds the path to the compile command using the following switch:

```
--precompiled=<path_to_pre-build_divine_libs>
```

The compilation of a program written in C/C++ programming language is invoked by the following command:

```
$ divine compile --llvm program.c --precompiled=<path>
```

The output file, `program.bc`, is then saved into the corresponding model directory.

Generating explicit state-space representation

Given an LLVM program, `DPtoolchain` calls DIVINE to generate the explicit graph representation of the program.

DIVINE generates the explicit state space and saves it into a DESS file by using the following command:

```
$ divine gen-explicit --probabilistic --compression program.bc
```

The `--probabilistic` switch tells DIVINE to store the transition probabilities in the DESS file. The use of `--compression` switch sets DIVINE to use lossless state space compression based on tree compression, which is especially efficient on large models. [11] As a consequence, the time for model construction and model checking process in PRISM is reduced.

As an output of this operation, a DESS file is created and saved in an appropriate model directory.

Translating model into states and transitions files for PRISM

`DPtoolchain` uses the `modeltranslator` program designed particularly for generating states and transitions files in format accepted by PRISM and for translating a LTL formula into PRISM syntax.

`modeltranslator` reads the DESS file corresponding to the model being verified, draws all necessary information from the Markov Decision Process and creates the states and transitions files which are then used as an input for PRISM model checker. The exact format of these files and their creation is more closely described in Section 3.3.4

Translating LTL formula

`modeltranslator` takes the LTL formula and uses the DIVINE LTL parser and textual substitution in order to create a corresponding LTL formula acceptable by PRISM. `modeltranslator` returns two LTL formulas, one asking about the minimal probability and the other asking about the maximal probability of the property being satisfied in the given model.

Probabilistic model checking in PRISM

The last step of the `DPtoolchain` is to run probabilistic verification inside the PRISM model checker. We provide PRISM with the created states and transitions files, `program.sta` and `program.tra`, which PRISM uses to build a model, and with two LTL properties to be verified saved in `prop` variable.

The command for running PRISMis the following:

```
$ prism -cuddmaxmem (4 * 1024 * 1024) -importstates program.sta
-importtrans program.tra -pf prop
```

During computation, PRISM relies on the CUDD (Colorado University Decision Diagram) package [14] which provides, among others, functions to manipulate Binary Decision Diagrams (BDDs) and Multi-Terminal-BDDs (MTBDDs), having both C and C++ interfaces [15]. However, by default, the upper memory limit of CUDD is 200 MB which may cause PRISM to run out of memory during model checking. Therefore, using the `-cuddmaxmem val` switch allows us to set the upper memory limit to a higher value (in KB), in case a machine that is being worked on has significantly more memory. In our case, we set the upper memory limit to `cuddmaxmem (4 * 1024 * 1024)`, that is 4 GB.

5 Evaluation

In this chapter, we present several examples covering different domains in order to evaluate efficiency of the realised tool chain.

Generally, we can observe two possible approaches to using C/C++ programming language for probabilistic model analysis. The first approach uses C/C++ for modelling various randomised algorithms and systems exhibiting probabilistic behaviours whilst in the second approach, real C/C++ programs are analysed using probability.

5.1 Case Studies

In this section, we will present each of the studied examples. The first example, the Sock Selection Problem, focuses on the use of probability in a randomised algorithm implemented by a single-threaded C program. The second problem, the Thief Problem, serves to demonstrate probabilistic analysis on a multi-threaded program using C++11 thread library, it illustrates the use of DIVINE's masking functions described in Section 3.3.2 and presents the use of an atomic proposition `joined` in order to consider only fair runs during model checking. The last example, Fischer Mutual Exclusion Protocol, presents a real multi-threaded protocol implemented in C programming language, and the use of an LTL fairness assumption.

5.1.1 Sock Selection Problem

To begin with the simplest of the problems, we present the randomised Sock Selection Problem [16] written in C programming language, whose main purpose is to demonstrate the implementation of probabilistic behaviour using the `__divine_choice` built-in function.

The Sock Selection Problem models an algorithm for finding a matching pair of socks. The problem is described as follows. A person successively draws socks from a drawer which contains socks of n colours. There is a limitation that the person can hold only two socks at a time. In case the person does not hold a matching pair, he randomly drops one of the two socks and draws the next sock from the drawer. This drop-and-draw process takes place until the person finds a matching pair or until there are no

socks left in the drawer. Each draw of a sock of a specific colour happens with certain probability derived from the ratio of colours.

We have implemented this algorithm for the number of colours $n = 2$ and $n = 3$. The corresponding programs are `draw_two_socks_randomized.c` and `draw_three_socks_randomized.c`

In the algorithm, the `__divine_choice` function is used three times. Firstly, it serves to decide what colour the currently drawn sock will be, depending on the rate of colours in the sock drawer.

Secondly, it simulates coin flipping in order to decide which sock to drop in case the person holds two non-matching socks. In this case, we use `__divine_choice(2,1,1)` to obtain two equiprobable choices creating two branches in MDP. One branch corresponds to dropping a sock in the left hand, the other one to dropping the sock in the right hand.

The last use of the `__divine_choice` is at the beginning of the algorithm. It is used to determine the ratio of the first colour. The ratio can be set to any of the following values (with the same probability): $3/10, 4/10, 5/10, 6/10, 7/10$. The ratio of other colours is derived from the probability of the first color, the probability is uniformly distributed among the remaining colours. The following line of code shows setting of the probability of the first colour:

```
26    int prob = (__divine_choice(5,1,1,1,1,1)) + 3;
```

In case we consider just two colours of socks, the drawing of socks is executed by running the following code:

```
61    switch (__divine_choice(2, prob, 10 - prob)) {
62    case 0: \\drawn a white sock
      ⋮
66    case 1: \\drawn a black sock
      ⋮
```

However, for the computations presented in Table 5.2, we omit the third probabilistic choice which sets the ratio of the first colour, and by extension, of the remaining colors. Instead, we set the ratios manually in order to obtain reasonably understandable and easily computable results.

The studied probabilities of this model are of the following kind. Given the ratio of white to black socks 9:1; find out the probability that the eventually found matching pair of socks will be black, find out the probability that the first two drawn socks will be of the same colour, or find out the probability that a matching pair will not be found.

In order to be able to study properties of this kind, we introduced atomic propositions `found`, to indicate that a matching pair was found, `lucky`, saying that the first two drawn socks were matching, and `black` and `white`, stating what the colour of the found matching pair was. For the result of verification of various properties, see Table 5.2.

5.1.2 Thief Problem

As the significant advantage of model checking lies in the verification of concurrent systems, in the following examples we will consider multi-threaded programs. The first example, the Thief Problem, is a simple multi-threaded program that does not use any synchronisation concepts such as mutexes, conditional variables, etc. It serves not only to show the successfully realized probabilistic model checking of C++ programs implementing threads from C++11 THREAD library, but also to demonstrate the way of ensuring fairness and the impact of DIVINE built-in functions `__divine_interrupt_mask` and `__divine_interrupt_unmask`. As we have seen in Section 3.3.2, they allow the user to define atomic sections over a sequence of instructions, thus, to significantly reduce the state space of a model.

The Thief Problem models a situation in which a thief tries to steal a diamond from a guarded museum room. The thief and a guard are each represented by a thread. The museum room is described using a simplified discrete representation.

The guard protects the displayed diamond placed in the middle of a rectangular museum room. He walks from one side of the room to the other, following the same line every time. To steal the diamond, the thief needs to cross the line followed by the guard. However, stealing the diamond is not so easy. The complication is that at any point the guard can decide to change the direction and start walking the opposite way.

The thief follows a simple strategy. He waits until the guard's back is turned to the diamond and the guard is walking away from the place where

the thief is hidden. At such moment, the thief rushes to the diamond, picks it up and runs away, crossing the guard's walking line again. If the guard turns during the thief's run and sees the thief, we claim he has caught the thief.

In our implementation, the line followed by the guard is divided into 11 sections. Each section is assigned two values, `turnL` and `turnR`. These values describe the probability of the guard changing his walking direction. The probability of the guard changing his direction increases with the decreasing distance of a wall he is walking toward.

Another possible employment of probability is to include a decision before every guard's step. The guard decides whether to make a step or stay at his current position. This decision can be simulated by coin flipping.

As concurrent systems are modelled by means of interleaving, the state space of the Thief Problem grows significantly. However, it is possible to reduce the size of the corresponding MDP without influencing the result of the verification by using DIVINE's functions `__divine_interrupt_mask` and `__divine_interrupt_unmask`, i.e. by limiting the number of interleavings.

Thanks to a convenient use of these built-in function at various parts of the code, the state space of the Thief Problem can be reduced of 33.5%.

The model introduces two straightforward atomic propositions, `caught` and `stolen`, and an atomic proposition `pickedup` to indicate that the thief picked up the diamond. In order to ensure that the model checking considers only realistic paths, a special atomic proposition is used, called `joined`. It is present only after both thief and guard threads have finished. Therefore, the used LTL fairness assumption is $F(joined)$.

In this model, we measured the probability of the thief stealing the diamond, or of the guard catching the thief either before or after picking up the diamond.

5.1.3 Fischer's Mutual Exclusion Protocol

The last case study is an example from DIVINE's collection of examples, the Fischer's Mutual Exclusion Protocol, programmed in C using PTHREAD library. This example is used for demonstration of the verification of the safety and liveness properties, which are defined inside the source code.

Fischer's mutual exclusion protocol is a well known delay protocol that ensures mutual exclusion among N processes using real-time clocks and a shared variable `lock`.

The algorithm uses a shared variable `owner` to remember which process requested access to the shared resource. In case `owner = 0`, it indicates that the `lock` is available. If a process reads `owner = 0`, it writes its process id to the variable and delays for some time. In case the value of `owner` still equals the process' id after the delay, the process enters the critical section.

In order to study the safety and liveness properties, we have introduced atomic propositions `waitX`, indicating that a thread X has requested the lock and is waiting to obtain it, and `criticalXin` and `criticalXout`, indicating that a thread X entered or left its critical section. In our implementation, the protocol uses two threads trying to enter the critical section. Thus, the exclusion property is defined as:

```
G((critical0in → (!(critical1in) W critical0out)) &&
   (critical1in → (! critical0in W critical1out)))
```

and the progress property:

```
G(wait0 → F(critical0in)) && G(wait1 → F(critical1in)).
```

5.2 Measurements

The Table 5.2 contains experimental values of probabilistic model checking executed on programs described in the previous section and on the two programs presented in Section 3.2, in Figures 3.3 and 3.4.

The second column of Table 5.2 shows the time spent on compilation and generation of explicit state space of each program using DIVINE. The number of states of the generated state space is presented in the next column, MDP size. Columns PRISM P_{min} and PRISM P_{max} present results of the quantitative model checking. The three following columns refer to duration of model construction and model checking in PRISM. The last column sums the duration of the whole `DPtoolchain` execution.

Model	Compilation / generating MDP (s)	MDP size	LTLproperty	PRISM P_{min}	PRISM P_{max}	PRISM time (s) model build	PRISM time (s) MC P_{min}	PRISM time (s) MC P_{max}	Total time
Two Socks Problem	1.2 / 0.6	122	$F(lucky)$	0.82	0.82	0.011	0.0	0.003	3.6s
			$F(lucky)\&\&F(white)$	0.81	0.81	0.011	0.013	0.108	3.6s
			$F(lucky)\&\&F(black)$	0.01	0.01	0.011	0.013	0.039	3.6s
			$F(white)\&\&G(!(lucky))$	0.151875	0.151875	0.054	0.026	0.044	3.7s
			$F(white)$	0.961875	0.961875	0.055	0.002	0.006	3.6s
			$F(black)$	0.026875	0.026875	0.054	0.002	0.007	3.6s
			$G(!(found))$	0.01125	0.01125	0.054	0.006	0.005	3.6s
Thief Problem	2.3 / 132.1	64 385	$F(caught)$	0.0	fail[1]	332.1	1.65	149.8	10m21s
			$F(caught), fair : F(joined)$	0.0	fail[1]	332.1	74.8	678.2	20m22s
			$F(stolen), fair : F(joined)$	0.0	1.0	370.4	202.4	643.1	22m34s
Toy 0	1.3 / 0.6	37	$F(assert)$	0.75	0.875	0.018	0.005	0.001	3.7s
Toy 1	4.6 / 0.8	79	$F(joined)$	0.0	1.0	0.05	0.0	0.004	7.7s
			$F(assert)$	0.0	0.5	0.037	0.001	0.005	6.0s
			$F(assert), fair : F(joined)$	0.5	0.5	0.045	0.12	0.029	3.9s
Fischer Protocol [2]	2.3 / 32.2	80 793	$progress$	0.0	1.0	226.2	2125.1	8594.3	183m
			$progress, fair : F(joined)$	1.0	1.0	240.0	110.5	262.5	11m1s
			$exclusion$	1.0	1.0	296.8	1354.4	4187.6	98m34s
Fischer Protocol (with bug)	1.3 / 52.3	137 345	$exclusion$	0.0	1.0	375.5	4602.5	11926.6	282m44s
			$exclusion, fair : F(joined)$	0	fail[1]	513.4	6745.1	x	127m4s
			$progress$	0.0	1.0	452.4	2393.3	16532.5	324m2s
			$progress, fair : F(joined)$	1.0	1.0	465.2	189.0	531.5	20m47s

Table 5.1: Measurements

1. PRISM failed to provide a valid result or crashed on CUDD error 2. LTL formulas for progress and exclusion properties are introduced in Section 5.1.3

For these experiments, we used the current stable version of PRISM, 4.2.1, and DIVINE version 3.3.1.

The experiments were performed on models that could be considered as small. We can observe that compilation and generation of the state space in DIVINE takes approximately a proportional amount of time to the size of a model. While, on the side of PRISM, the time of the model verification varies a lot for different LTL formulas, and for larger models, the time can exceed several hours.

During these experiments, we discovered several bugs in the PRISM model checker and discussed them with the PRISM developers. Generally, we have discovered two types of bugs.

The first bug occurred with some correctly constructed LTL formulas which caused PRISM to crash on CUDD error. This happened in the case of the Sock Selection Problem and Fischer's Mutual Exclusion Protocol. When we notified PRISM developers, we were informed that this bug had been fixed in the current development version (4.2.1.dev.r9678). The CUDD error supposedly happened either because of running out of CUDD memory, or for an unknown reason. In our case, it seemed that the memory wasn't the issue. However, when we started using the current development version, it caused other problems. The two problems we have encountered were that PRISM computed incorrect results of probabilistic analysis, and it returned various results for identical formulas depending on whether only a single formula was verified or whether PRISM was provided with a list of formulas. Therefore, we have returned back to using the current stable version.

The second type of bug consisted of PRISM returning an `Infinity` and `-Infinity` probability values. This problem happened, for example, during verification of the Thief Problem for the presence of an atomic preposition `caught`, as you can see in Table 5.2.

6 Conclusion

The aim of this work was to create a tool chain connecting DIVINE and PRISM model checkers in order to enable quantitative verification of probabilistic systems modelled in the C/C++ programming language with the probabilistic choice operator.

The main issue of the implementation was to create an explicit model in DIVINE containing necessary information for probabilistic analysis, and to be then transformed into a model format acceptable by PRISM. For the purpose of the model transformation, a special program, `modeltranslator`, was implemented. `modeltranslator` takes care of creating the necessary inputs for probabilistic model checking in PRISM. The overall functionality of probabilistic model checking is ensured by the `DPtoolchain` script which connects the two model checkers and the `modeltranslator` program and processes their input and output files.

Despite the fact that the chaining of DIVINE and PRISM was successfully implemented, the tool chain cannot currently guarantee to always return correct results due to several bugs that we have discovered in PRISM. These problems have been brought to attention of the PRISM developers and should be soon fixed. During implementation of `DPtoolchain`, several bugs were also discovered in the DIVINE model checker. These were immediately fixed.

This thesis presents a set of simple models created for two purposes. Firstly, it is used to demonstrate how probability can be introduced in C/C++ programs using the DIVINE built-in probabilistic choice operator and atomic propositions, and how to create atomic sections of code. Secondly, experimental evaluation of the approach to probabilistic model checking adopted in this thesis was performed on these models.

At this moment, the bottleneck of the tool chain is the time spent on model construction and verification in the PRISM model checker. This could be partly anticipated since PRISM's primary interest does not lie in the verification of imported explicit models, but in the symbolic verification of models described using the PRISM language. However, we believe that the time for verification, which is already reduced by using DIVINE's state space reduction techniques, could be further shortened by a reduction of an MDP with respect to verified LTL formulas.

Further possible work to be carried out consists of saving state flags of an MDP in a PRISM label file (`*.lab`) instead of a PRISM state file (`*.sta`), once PRISM will have finished its implementation. Since the label file stores for each atomic proposition only a list of states in which it is present, it would considerably reduce the size of explicit models passed to PRISM. Moreover, it could significantly accelerate the model construction and analysis in PRISM as, instead of accessing all states to find out whether a certain atomic proposition is present in any state, it could directly access only those states where it is present.

Bibliography

[1] E. A. Emerson, "The beginning of model checking: A personal perspective," in *25 Years of Model Checking*, O. Grumberg and H. Veith, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 27–45. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69850-0_2

[2] O. Grumberg and H. Veith, Eds., *25 Years of Model Checking: History, Achievements, Perspectives*. Berlin, Heidelberg: Springer-Verlag, 2008.

[3] E. Clarke, "The birth of model checking," in *25 Years of Model Checking*, ser. Lecture Notes in Computer Science, O. Grumberg and H. Veith, Eds. Springer Berlin Heidelberg, 2008, vol. 5000, pp. 1–26. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69850-0_1

[4] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] "DIVINE: Model Checking for Everyone," accessed: 2015-11-16. [Online]. Available: http://divine.fi.muni.cz/

[6] "PRISM - Probabilistic Symbolic Model Checker," accessed: 2015-03-06. [Online]. Available: http://www.prismmodelchecker.org/

[7] J. Tůmová, "Quantitative linear-time model checking [online]," 2010 [cit. 2015-05-17]. [Online]. Available: http://is.muni.cz/th/98614/fi_r/

[8] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[9] C. Baier, "On algorithmic verification methods for probabilistic systems," 1998, habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim.

[10] J. Barnat, I. Černá, P. Ročkai, V. Štill, and K. Zákopčanová, "On Verifying C and C++ Programs with Probabilities," 2015.

[11] "Divine: Model checking for everyone," accessed: 2015-04-07. [Online]. Available: http://divine.fi.muni.cz/manual.html

[12] P. Ročkai, "Model checking software [online]," Ph.D. Thesis, Masarykova univerzita, Fakulta informatiky. [Online]. Available: http://is.muni.cz/th/139761/fi_d/

[13] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification (CAV 2013)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 863–868.

[14] "PRISM Manual | Main / PTAs," accessed: 2015-03-16. [Online]. Available: http://www.prismmodelchecker.org/manual

[15] "Introduction," accessed: 2015-04-12. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/node1.html

[16] "Sequential randomized algorithm - the sock selection problem (input randomization)," accessed: 2015-03-18. [Online]. Available: http://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/randomized-algorithms/examples/sequential.html