TU Clausthal
Clausthal University of Technology

Modellierung und Formale Verifikation von Reaktiven Systemen am Beispiel einer Fahrzeugfunktion

Research Track Paper

10. Juli 2019

Institut
Institute for Software and Systems Engineering

Gutachter
Prof. Dr. Andreas Rausch
Prof. Dr. Jürgen Dix

Betreuerin
Frau M.Sc. Adina Aniculăesei

Verfasser
Jan Toennemann
Matrikelnummer 444657

Studiengang
Informatik

Fakultät
Fakultät III
Mathematik/Informatik und Maschinenbau

Erklärung der Urheberschaft

Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass alle Stellen dieser Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsstelle vorgelegt wurde.

Des Weiteren erkläre ich, dass ich mit der öffentlichen Bereitstellung meiner Arbeit in der Instituts- und/oder Universitätsbibliothek einverstanden bin.

Clausthal-Zellerfeld, 10. Juli 2019	
Ort, Datum	Unterschrift

Contents

Abstract

In the age of self-driving cars and other growing uses of autonomous systems, ensuring that safety-critical software works correctly became even more important. The reactive systems that are the core of this change are becoming increasingly complex and tend to work not only with reliable input from inside the system, but also incorporate measured data with a possible error and received data with volatile reliability.
In this paper we will explore several different approaches to applying formal methods to verify properties over such complex reactive systems. We will compare the synchronous data-flow programming language Lustre and the accompanying Kind 2 model checker, the communication-focused formalism mCRL2 and the probabilistic model checkers PRISM and Storm with regard to a case study of a reactive system exhibiting such properties, based on an industrial automotive function.
The case study consists of a model with several features we have observed in real-world examples, e.g. heavy use of floating point numbers, error models for input data, stochastically distributed input data and dependence on both current and past system states. Requirements imposed upon the model are also modeled after real-world prototypes and are defined in a way that allows checking for their satisfiability in all three different model implementations.
Used languages and toolsets are compared with regard to applicability given the case study. The strengths and weaknesses of the different approaches are discussed to give a rough overview of the current state of model-checking complex reactive systems.

1. Introduction

Reactive systems and requirements defined upon them are getting increasingly complex. These systems, used to build a variety of applications, such as in multimedia devices or avionic systems, exhibit stochastic behaviour and also operate under constraints on timing and other resources [20]. Ensuring the correctness of these systems is of paramount importance, especially for those systems deployed in safety-critical applications.

Through their continuous interaction with their operation environment, reactive systems are subject to a variety of external stimuli. This heterogeneity raises the pressure on verification and validation (V&V) techniques, which are used to ensure the correctness of reactive systems. Thus, V&V approaches used for the verification of reactive systems are required to achieve a higher level of flexibility in handling heterogeneous environment input. Looking at the interaction between a reactive system and its operation environment, we see two fundamentally different entities, each of them working under their own set of rules. On one hand, the processes in the operation environment are subject to the laws of physics and take place in continuous time. On the other hand, the reactive system takes discrete values as input and computes discrete values in a discrete time model.

1.1. Motivation

In this paper we are examining state-of-the-art verification techniques for reactive systems which depend on several types of input from its surroundings. The contrast between the actual physical environment and the discrete model present several challenges; we need to describe a system which exhibits the following properties:

- computations with floating point numbers,
- some non-exact inputs (e.g. because they are measured) such that error ranges (absolute and/or percental) need to be considered,
- some inputs prevent best-/worst-case verification of the model such that a distribution for these parameters needs to be considered and an approach like quantitative verification is required, and
- not only working on direct input, but keeping an internal state and saving data, such that the implemented model is required to preserve values of the state variables over several iterations.

Initially, we intended to verify an automotive function which exhibited all these features and had a very complex overall model. Due to the complexity of this function, we chose to focus on a simplified mathmatical function, which still features all these properties. We verify it using several different approaches, with the explicit goal to choose the most suited one for verification of the initial automotive function.

1.2. Research Questions

To progress with solving our problem, we had to answer the following questions:
How can a complex reactive system under consideration of

1. measurement errors,
2. stochastic parameters, and
3. past system states

be modeled?
How can a complex reactive system unser consideration of the

1. given assumptions and conditions,
2. dependency between current system state and past system states,
3. stochastic parameters of the function, and
4. desired confidence level for the requirement satifiability

be verified?

We will compare three different approaches to answer these questions, using the synchronous data-flow programming language Lustre and the accompanying KIND 2 model checker, the communication-focused formalism MCRL2 and the probabilistic model checkers PRISM and STORM.

1.3. Related Work

In the recent few decades, model-checking has increasingly been used to verify the correct behaviour of reactive systems[13, 20, 25]. A structured approach to chose among modeling languages and tools for the formal analysis of cyber-physical systems is shown in [3]. It takes into account three elements: viewpoints, which reflect the stakeholders' concerns, mathematical formalisms – needed to model the stakeholders' interests – and tools which implement these formalisms in their respective input languages. Recognizing the diversity of verification and validation approaches for reactive systems, the RERS challenge [17] provides a forum for experimental profile evaluation based on specifically designed verification tasks. The benchmarks are synthesized to exhibit increasingly complex properties, such as safety or liveness, reactive systems varying from a few hundred lines to million hundred lines of code, as well as language features such as assignments or pointer arithnetics. However, these challenges have focused only on functional properties, leaving for future research issues such as stochastic behaviour and errors in the measured sensor data.

The tools selected for our survey have found various applications in the past. LUSTRE and KIND 2 have also been used to verify SIMULINK models, e.g. of aviation controls[1] or of a triplex sensor voter[12]. MCRL2 has been used to verify parts of the software controlling the Large Hadron Collider at CERN[18, 24]. There already are numerous approaches detailing the use of the PRISM model checker for safety-critical systems where some kind of uncertainty is involved[14, 22]. In [5], the authors demonstrate the verification of properties which must hold within a given confidence interval using PRISM.

1.4. Structure

Following these introductory sections, Chapter 2 will be used to introduce necessary basics in model checking, stochastic verification and the used languages and tools. Subsequently, in Chapter 3 the main concept of this paper is detailed, explaining the reasoning behind the case study and our choice of tools. We will give an overview of the properties that our model possesses and the derivated requirements for the languages and toolchains used.

We will then introduce the case study in Chapter 4, based on an existing function. A mathematical model for the case study is defined, over which we will impose requirements that are to be verified. The defined model, even though heavily simplified compared to the original automotive function, still retains the properties we identified to be very challenging to verify.

In Chapter 5 we will implement the models in the three chosen tools and will perform the verification itself, listing both the queries and the verification results of the verifiable queries per tool. A summarized overview about the tools and their actual features is given. These results are then discussed in Chapter 6, where we detail the strengths and weaknesses of the surveyed tools with regard to our model. We discuss complexity of both models and the actual verification and techniques to reduce the generated state space, ending the chapter with a conclusion about the current state of model checking with regard to growing system complexity.

2. Preliminaries

In this chapter we will introduce concepts that might not be known to the reader of this paper, specifically the basics of stochastic verification, the *truncated normal distribution* used later in the mathematical model of the case study in Chapter 4 as well as the basics of the languages and tools used to implement and verify the model in Chapter 5.

2.1. Stochastic Verification

While labelled transitions systems and finite-state machines are already considered to be part of Computer Science basics, several concepts used in stochastic verification are not yet well-known. In this section, we will introduce *Discrete-Time Markov Chains* as well as *Markov Decision Processes*, which extend the concept of transition systems by properties required for stochastic verification.

The definitions in Section 2.1.1 and Section 2.1.2 are taken from the book 'Principles of Model Checking'[7] and have only been slightly adapted. For a more in-depth introduction, the reader is referred to this book.

2.1.1. Discrete Time Markov Chains (DTMC)

Discrete Time Markov Chain are transition systems where there are no nondeterministic choices, but successor states may be chosen by probabilistic choices. This way, the successor of the current state is chosen by a probability distribution depending only on the current state. The choices depend only on the current state and not on any past ones, such that DTMCs are not affected by the history of past chosen states.

We consider a *Discrete Time Markov Chain* to be a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{\text{init}}, AP, L)$ where

- S is a countable, non-empty set of states,
- $\mathbf{P} : S \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$:

$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$

- $\iota_{\text{init}} : S \to [0, 1]$ is the *initial distribution* such that $\sum_{s \in S} \iota_{\text{init}}(s) = 1$, and
- AP is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

A simple example of such a DTMC is given in Example 2.1 and visualized in Figure 2.1. The values of the probability transition function are given for each enabled transition (where $\mathbf{P}(s, s') \neq 0$ for $s, s' \in S$). This DTMC is implemented in PRISM code in Section 2.3.3.

2.1.2. Markov Decision Processes (MDP)

Markov Decision Processes re-introduce the concept of nondeterminism into probabilistic transition systems, since they permit both probabilistic and nondeterministic choices. This makes them

$S = \{a, b, c, d, e\}, \quad AP = S, L(s) = \{s\}, \quad \iota_{\text{init}}(a) = 1,\ \iota_{\text{init}}(s) = 0 \forall s \in S \setminus a,$
$\mathbf{P}(a, b) = 0.5,\ \mathbf{P}(a, c) = 0.5,\ \mathbf{P}(b, d) = 0.8,\ \mathbf{P}(b, e) = 0.2,\ \mathbf{P}(c, d) = 1,\ \mathbf{P}(d, e) = 1$ and
$\mathbf{P}(s, s') = 0$ for any other $s, s' \in S$ not explicitly listed

Example 2.1: Example of a Discrete Time Markov Chain

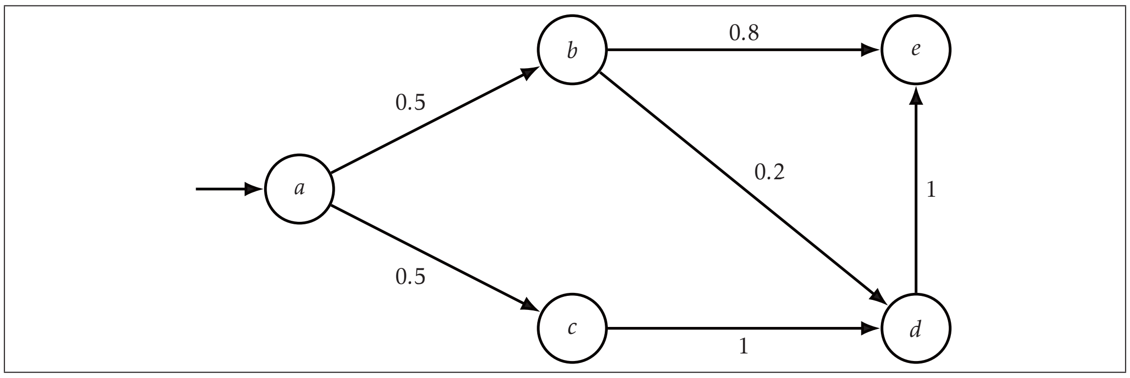

Figure 2.1.: Visualization of Example 2.1

especially useful when parts of the model can be quantified and estimated while others cannot, such that probabilities for certain, but not all, choices can be given. The nondeterministic choices can be used for the parts of the model that cannot be estimated or when all possible outcomes need to be guaranteed.

We consider a *Markov Decision Process (MDP)* to be a tuple $\mathcal{M} = (S, Act, \mathbf{P}, \iota_{\text{init}}, AP, L)$ where

- S is a countable set of states,
- Act is a set of actions,
- $\mathbf{P} : S \times Act \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$ and actions $\alpha \in Act$:
$$\sum_{s' \in S} \mathbf{P}(s, \alpha, s') \in \{0, 1\},$$
- $\iota_{\text{init}} : S \to [0, 1]$ is the *initial distribution* such that $\sum_{s \in S} \iota_{\text{init}}(s) = 1$, and
- AP is a set of atomic propositions and $L : S \to 2^{AP}$ a labeling function.

An action α is *enabled* in state s if and only if $\sum_{s' \in S} \mathbf{P}(s, \alpha, s') = 1$. Let $Act(s)$ denote the set of enabled actions in s. For any state $s \in S$, it is required that $Act(s) \neq \emptyset$. Each state s' for which $\mathbf{P}(s, \alpha, s') > 0$ is called an *α-successor* of s.

In accordance with this definition, Example 2.2 gives an MDP based on Example 2.1, extended by actions and nondeterministic transitions outgoing from state b. Note that transitions are nondeterministic if there are multiple outgoing transitions from a state with different actions, where for each action the sum of probabilities sums to one. A visual representation of this example is shown in Figure 2.2 and the example is implemented in Section 2.3.3 as well.

$S = \{a,b,c,d,e\}$, $AP = S$, $L(s) = \{s\}$, $\iota_{\text{init}}(a) = 1, \iota_{\text{init}}(s) = 0 \forall s \in S \setminus a$, $Act = \{\alpha,\beta,\gamma,\epsilon\}$,
$\mathbf{P}(a,\alpha,b) = 0.5$, $\mathbf{P}(a,\alpha,c) = 0.5$, $\mathbf{P}(b,\beta,d) = 1$, $\mathbf{P}(b,\gamma,e) = 1$, $\mathbf{P}(c,\beta,d) = 1$,
$\mathbf{P}(d,\gamma,e) = 1$, $\mathbf{P}(e,\epsilon,e) = 1$ and $\mathbf{P}(s,\epsilon,s') = 0$ for any other $s,s' \in S$ not explicitly listed

Example 2.2: Example of a Markov Decision Process, adaptation of Example 2.1 featuring nondeterminism

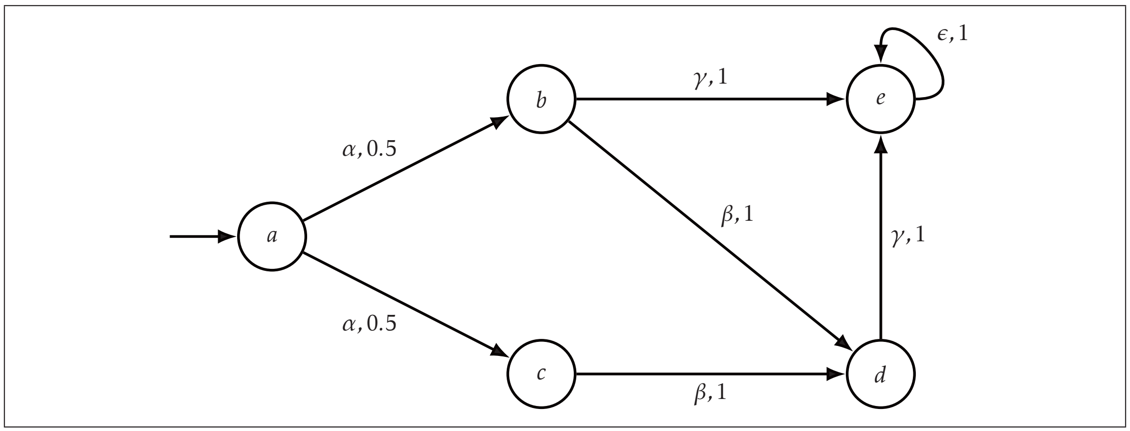
$$\begin{tikzcd}
 & b \arrow[r, "{\gamma,1}"] \arrow[drr, "{\beta,1}"] & & e \arrow[loop right, "{\epsilon,1}"] \\
\arrow[r] & a \arrow[u, "{\alpha,0.5}"] \arrow[d, "{\alpha,0.5}"'] & & \\
 & c \arrow[rr, "{\beta,1}"'] & & d \arrow[uu, "{\gamma,1}"']
\end{tikzcd}$$

Figure 2.2.: Visualization of Example 2.2

2.2. Truncated Normal Distribution

To estimate the distribution of input parameters, we will use a doubly truncated normal distribution. Such a distribution is similar to a normal distribution in that it is distributed around a mean with a standard deviation, but its lower and upper limits can be chosen such that the probability distribution function between those two points sums up to 1.[19]

We consider a function to be the probability density function (PDF) of a truncated normal distribution if it can be expressed as

$$\frac{\phi(\xi)}{\sigma\left(\Phi(\beta) - \Phi(\alpha)\right)}$$

with parameters

$$\alpha = \frac{a-\mu}{\sigma}, \quad \beta = \frac{b-\mu}{\sigma}, \quad \xi = \frac{x-\mu}{\sigma}$$

and functions

$$\phi(\xi) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}\xi^2\right), \quad \Phi(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right).$$

Here, the mean is defined by μ, the standard deviation by σ and the upper and lower truncation limit by a and b respectively. α, β and ξ have been introduced to shorten the function. The function erf is the Gauss error function defined as

$$\operatorname{erf} = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

2.3. Used Tools & Languages

In Section 1.2 we mentioned the tools we are going to use throughout this paper, the reasoning behind the choices are explained in Chapter 3. Here, we will give a short introduction into each language and tool to give the reader a basic understanding of the model building and verification process. For a more in-depth introduction into each tool, the reader is referred to the respective tools' website.

2.3.1. Lustre & Kind 2

Lustre[1] is a synchronous data-flow programming language, Kind 2[2] is an SMT-based model checker for synchronous reactive systems which gives counter-examples when a requirement is not met.[6] Models are implemented by defining a sequence of nodes, which take several input parameters, perform calculations and output results. These nodes can be chained to create sequential data flow, concurrency or nondeterminism are not supported. The language only supports a limited set of operators and no loops such that more complex operations cannot be implemented.

```
node Ex(a, b : real) returns (d : real);
(*@contract import ExSpec(a, b) returns (d); *)
var
    c : real;
let
    c = a + b;
    d = c + 0 -> pre(c);
tel
```

Listing 2.1.: Example of a Lustre computation node

The contract line in the node specification of Listing 2.2 is not part of standard Lustre syntax, but instead of Kind 2 syntax. Contracts are comparable to nodes, but instead of equation systems they contain model checking specifications like assumptions and guarantees. We use the contract to specify the limitations on the input parameters as well as expectations for the output parameters.

```
contract ExSpec(a, b : real) returns (d : real);
let
    assume a >= 0.0;
    assume b <= 2.0 * a;

    guarantee not (d >= 10 * a * b);

    mode ExMode (
        require c <= 0.0;

        ensure not (d > 0.0);
    );
tel
```

[1]http://www-verimag.imag.fr/The-Lustre-Toolbox.html
[2]http://kind.cs.uiowa.edu/

```
node Ex(a, b : real) returns (d : real);
(*@contract import ExSpec(a, b) returns (d); *)
var
    ...
tel
```

Listing 2.2.: Example of a Kind 2 contract for the node in Listing 2.1

2.3.2. mCRL2

mCRL2[3] is a formalism 'which extends the algebra of communicating processes (ACP) [...] with various features including notions of data, time, and multi-actions'[16]. It has been used for the verification of a variety of reactive systems and does natively support probability distributions[4].

In mCRL2, processes are defined as transition systems with parametrizable actions. These actions – if they have parameters – can be assigned either numeric constants or one can use the `sum`-operator to enumerate over a limited domain. Listing 2.3 shows an mCRL2 system specification with an unparametrized action `a`, a parametrized action `b` taking real numbers as parameter and the parametrized action `c` which expects a natural number as parameter. The process `P` combines all of these, transitioning out of the initial state using action `a`, transitioning out of the reached state using action `b` with the constant parameter `5/2` and then using the `sum`-operator to enable transitions using actions `c` for the parameters in the range $[0, 10] \in \mathbb{N}$ to the last state of the model, then starting process `P` again, transitioning into the initial state.

```
act a;
act b: Real;
act c: Nat;

proc P = a . b(5/2) . sum n: Nat . (n >= 0 && n <= 10) -> c(n) . P;

init P;
```

Listing 2.3.: Example of an mCRL2 specification

The mCRL2 toolset is quite comprehensive and contains various tools, including the *LTSgraph* utility. Figure 2.3 shows the visualization of the Linear Transition System (LTS) using this tool, generated from the code in Listing 2.3. The effects of the bounded `sum`-operator can be seen in the amount of transitions from state 2 back to the initial state.

2.3.3. PRISM & Storm

PRISM[4] is a model checker that supports various types of probabilistic models, including the already introduced DTMCs and MDPs[20]. We will introduce the PRISM specification language here, which is also supported by the Storm[5] model checker, such that we can build probabilistic

[3] `https://www.mcrl2.org/`
[4] `http://www.prismmodelchecker.org/`
[5] `http://www.stormchecker.org/`

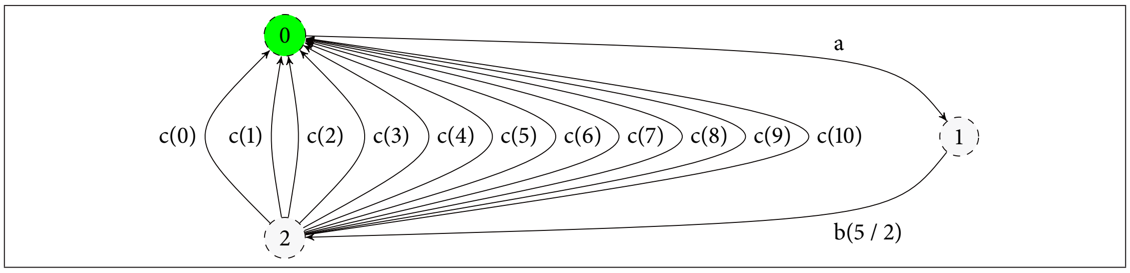

FIGURE 2.3.: Visualization of the LTS generated by Listing 2.3

models and apply quantitative verification in Chapter 5.

This specification language differs a bit from the other two since it is not explicitly designed to verify reactive systems. But since it natively supports stochastic verification, which the others do not, and is very flexible in what models can be created, we will attempt to use it to our purpose.

Listing 2.4 defines Example 2.1 using a PRISM specification. The very first line indicates the model type, here `dtmc` signals that the file contains a DTMC description. Computations are done in `modules`, this specification only contains the single module `exDTMC`, where the variable `s` is defined for the value range $[0, 4]$ and initialized with 0. Then, the transitions from Example 2.1 are defined; from state `s = 0` corresponding to a in the definitions, transitions to the states `s = 1` and `s = 2`, corresponding to b and c respectively, are defined, each with probability $\frac{1}{2}$. State `s = 3` matches state d in the definition, `s = 4` state e.

```
dtmc

module exDTMC
    s : [0 .. 4] init 0;

    [] (s = 0) -> 1/2 : (s' = 1) + 1/2 : (s' = 2);
    [] (s = 1) -> 4/5 : (s' = 4) + 1/5 : (s' = 3);
    [] (s = 2) -> 1 : (s' = 3);
    [] (s = 3) -> 1 : (s' = 4);
endmodule
```

LISTING 2.4.: Example from Section 2.1.1

Listing 2.5 contains the model file for Example 2.2, starting with the appropriate line indicating the `mdp` model type. The very first transitions are equal to that in Listing 2.4, all other transitions are defined without probabilities since these are either exactly 1 or nondeterministic.

```
mdp

module exMDP
    s : [0 .. 4] init 0;

    [] (s = 0) -> 1/2 : (s' = 1) + 1/2 : (s' = 2);
    [] (s = 1) -> (s' = 3);
    [] (s = 1) -> (s' = 4);
    [] (s = 2) -> (s' = 3);
    [] (s = 3) -> (s' = 4);
    [] (s = 4) -> (s' = 4);
endmodule
```

Listing 2.5.: Example from Section 2.1.2

3. Concept

Our research goal is to determine how to model a reactive system and its environment with regards to the requirements defined by the system properties as mentioned in Section 1.1. We had to work out which attributes were unique to this system and its environments and how they come into play when applying formal verification techniques. In addition, we had to deduce which criteria are important for the verification itself and how we could compare and evaluate these considering several different approaches.

We surveyed applicable languages and tools and decided to work with Lustre and KIND 2 - a synchronous data-flow programming language and an accompanying SMT-based model checker, mCRL2 - a formal specification language based on the algebra of communicating processes (ACP), as well as PRISM and STORM - two probabilistic model checkers, both using the same input format for the system model as well as for the requirement specification. Each tool proposes a different approach to specifying and model-checking reactive systems. We thus implemented the system model in various ways and compare both the results and the tools' respective applicability to our problem.

The reason to choose Lustre and KIND 2 resulted from the fact that it provided a native concept to implement iteration-based reactive systems and access data from previous iterations using the `pre`-operator. With the KIND 2 model checker, system behaviour and specification are closely coupled and in cases of violations, counter-examples are provided.

We chose mCRL2 since it is a process algebra explicitly designed for reactive systems and also has native support for probability distributions for input parameters. The toolset surrounding mCRL2 is quite powerful and features, in addition to the verification engine, also a visualization tool as well as a simulator.

To also perform quantitative model checking, we opted to use PRISM as a specification language since both its modeling and its verification capabilities appeared to be very comprehensive. After beginning to use PRISM, we also found the Storm model checker which conveniently uses PRISM models as an input format while providing better performance and additional features.

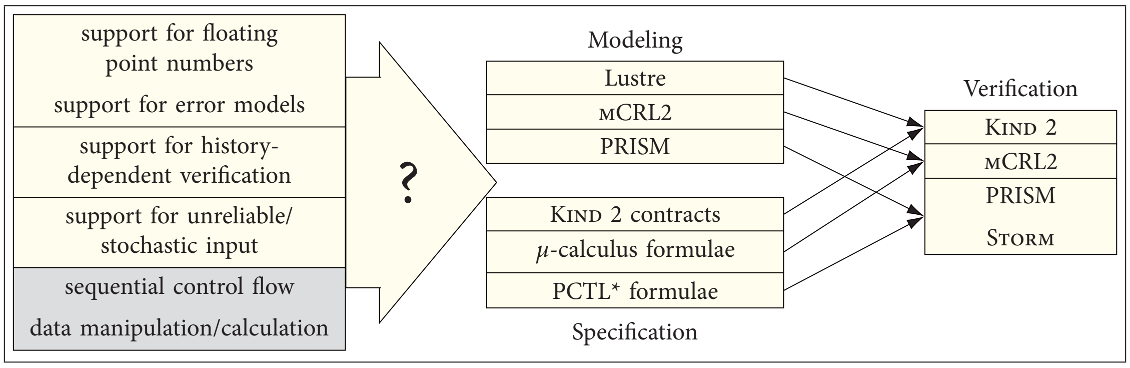

FIGURE 3.1.: Visualization of the model-building and verification concept

Figure 3.1 shows how we are going to approach this problem. We have selected different possible approaches and know how the verification step works, but we will need to see how many of our model properties can be actually be implemented in the different languages and tools.

The very basics of the model, the sequential control flow of the program and the ability to manipulate data and to calculate results, are supported by all of the tools. We also know that every approach has support for floating point numbers, but will need to ascertain to which extent they can be used in the verification process.

If the language supports floating point numbers in modeling and verification, we can assume that we are able to implement the error models quite as well, since effectively they just extend the range of given input data to include the uncertainty.

Support for history-dependent verification requires the ability to implement an iteration-based model and to save and load data across iterations; there are various ways to accomplish this, but in the end each approach will have its own way of implementing this.

We know that we can model probabilistic input parameters in PRISM models and that mCRL2 also has native support for probability distributions, but we will attempt to find ways to model our own probability distribution function in all three languages.

4. Case Study

To be able to compare the different toolsets with regard to applicability to the original automotive function, we perform a case study. The model built throughout this chapter preserves the main properties of the initial model as mentioned in Section 1.1 such that we can perform an evaluation to answer the question from Section 1.2.

4.1. Important Criteria

The function we base our simplified model on worked with measured values, but the final result was supposed to be verified considering the real values, which is why the possible deviation between measured and real values plays an important role here. The error models were given as tolerance intervals expressing either absolute or percental deviation. In our model we chose to represent this by using 'raw' (real) values as input and computing the function input parameters from these and the given error ranges.

In addition, the original function had several input parameters that were dependent on transmitted signals, which may be unavailable or less reliable at times. Both a best- and worst-case verification would not make sense for these signals, since in reality the reliability of these will be somewhere inbetween. This is the reason for a stochastically distributed input parameter in the model, as we have to assume a fixed distribution instead of relying on best- and worst-case-verification for this parameter. For the verification of a given function, the distribution should be estimated using tests and adjusted to include a reasonable buffer.

Model-checking 'is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.'[7] Although the input data for this model is limited by ranges, dealing with floating point numbers means we are still working with infinite domains. Choosing an appropriate discretization, we can obtain data from an enumerable finite domain, but are distancing ourselves from the model.

Considering that the model will be implemented in software and computers always work with discrete representations of floating point numbers, this can be mitigated by either

- knowing the target machine's exact discretization and applying this to the verifiable model implementation or
- choosing an appropriate discretization and shipping it with the verification results, instructing the developers to use this exact discretization when implementing the software system.

Having a stochastically distributed input parameter requires similar discretization work. While some of the tools we are using natively allow for stochastic verification to some extent, none support specifying an exact distribution function for a variable. Instead, we will have to decide on an appropriate discretization mechanism for the distribution function, which again distances us from the actual model.

4.2. Mathematical Model

The input parameters in our actual model represent measured values. To simulate this, the mathematical model uses the 'raw' input parameters $v_{1,\text{raw}}, v_{2,\text{raw}}$ such that by incorporating the error models we get the values v_1, v_2 emulating measured values. The error model e_1 represents an absolute error affecting v_1 additively as $e_1 \in [-e_{1,\max}, e_{1,\max}] \subset \mathbb{R}, e_{1,\max} \in \mathbb{R}_+$. Error e_2 shall affect v_2 multiplicatively as percental distortion within range $[-e_{2,\max}, e_{2,\max}] \subset \mathbb{R}, e_{2,\max} \in [0.00, 1.00] \subset \mathbb{R}_+$. Note that additionally, we do not want our values to go below 0, even under consideration of the error model. This is no problem with the percentual error, but we will need to put a constraint on the absolute error.

$$\begin{aligned}\text{value } v_{1,\text{raw}} \in [0, v_{1,\max}] \subset \mathbb{R}_+, v_1 = &\quad v_{1,\text{raw}} + e_1 \\ \Rightarrow &\quad v_1 \in [0, v_{1,\max} + e_{1,\max}] \subset \mathbb{R}_+\end{aligned}$$

$$\begin{aligned}\text{value } v_{2,\text{raw}} \in [0, v_{2,\max}] \subset \mathbb{R}_+, v_2 = &\quad v_{2,\text{raw}} \pm (e_2 \cdot 100)\% \\ = &\quad v_{2,\text{raw}} + v_{2,\text{raw}} \cdot e_2 \\ \Rightarrow &\quad v_2 \in [0, v_{2,\max} + v_{2,\max} \cdot e_{2,\max}] \subset \mathbb{R}_+\end{aligned}$$

Some of the parameters in the actual model cannot be realistically verified using worst-case or best-case analysis, e.g. the accuracy of GPS data. To handle these parameters, we will assume a stochastic distribution. The mathematical model includes a parameter weighting $w \in [0, 1]$ modeled as a *truncated normal distribution* as described in Section 2.2 with the mean at 0.75 and a standard deviation of 0.8, such that it is 'mostly close to 1'.

$$\text{PDF } f(x) = \frac{\dfrac{e^{-0.78125\cdot(x-0.75)^2}}{\sqrt{2\pi}}}{0.8\left(\dfrac{1}{2}\left(1+\operatorname{erf}\left(\dfrac{\frac{5}{16}}{\sqrt{2}}\right)\right)\right) - 0.8\left(\dfrac{1}{2}\left(1+\operatorname{erf}\left(\dfrac{\frac{-15}{16}}{\sqrt{2}}\right)\right)\right)}$$

$$= \quad 1.11208 \cdot e^{-0.78125\cdot(x-0.75)^2}, \quad 0 \le x \le 1$$

Since we are dealing with a continuous random variable, we cannot get the probabilities for specific events or points, but instead have to integrate the probability distribution function over an interval $[a, b]$ with $a, b \in [0, 1]$ to get the probability $p[a, b]$.

The model uses functions which save recent values and 'learn' from them, e.g. by building a moving average over the most recent x values. We define the `pre`-operator to access values from previous iterations, giving a placeholder value for the first iteration, such that $\texttt{pre}(v, 0)$ returns the value of v from the previous iteration and 0 in the first one. We will also allow them to be nested to access values from an arbitrary number of previous iterations.

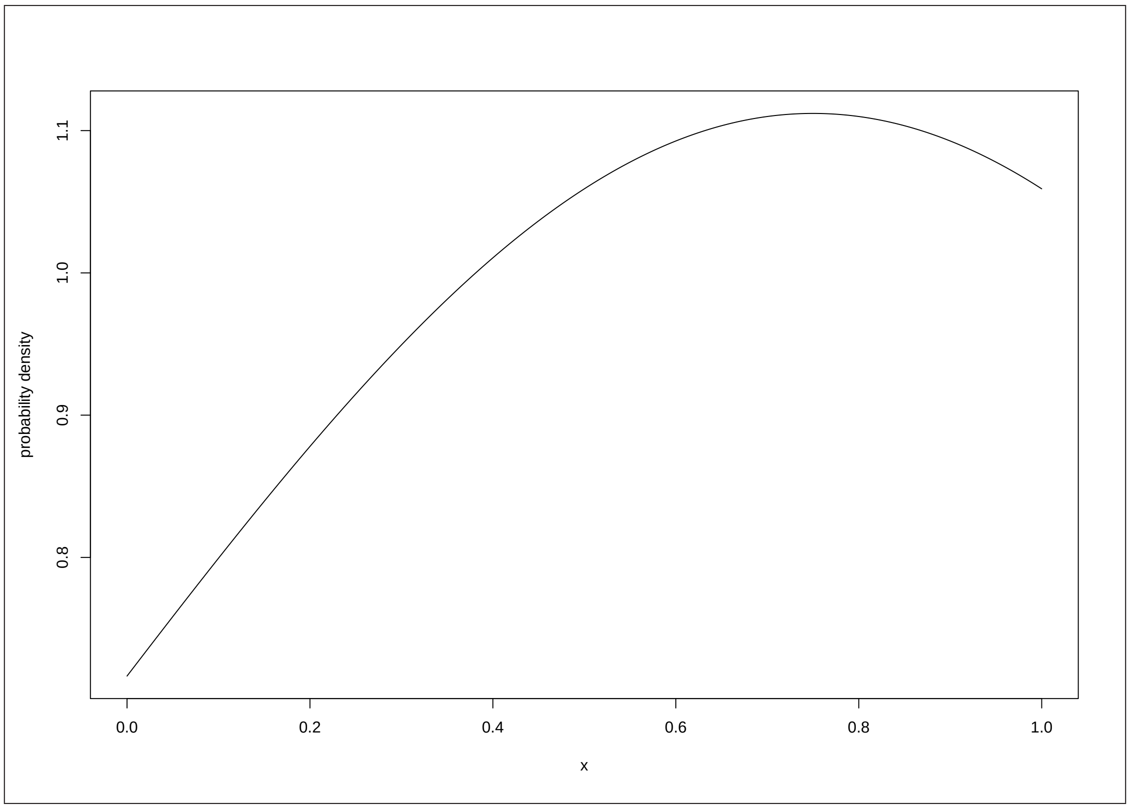

Figure 4.1.: Graph of the Probability Distribution Function for w

iteration	1	2	3	4	5	6
v	1	2	3	4	5	6
$\mathrm{pre}(v,0)$	0	1	2	3	4	5
$\mathrm{pre}(\mathrm{pre}(v,0),0)$	0	0	1	2	3	4

Table 4.1.: Example for the iteration-based use of the pre-operator

The results that are to be calculated are the following:

$$\begin{aligned} r &= v_1 + v_2 \\ r_w &= w \cdot r \\ r_p &= v_1 + \mathrm{pre}(v_2, 0) \end{aligned}$$

Assuming an iteration based model with discrete time $t \in \mathbb{N}$, where t is the iteration, we get the following formulas:

$$\begin{aligned} r(t \geq 0) &= v_1(t) + v_2(t) \\ r_w(t \geq 0) &= w(t) \cdot r(t) \\ r_p(t \geq 1) &= v_1(t) + v_2(t-1) \\ r_p(t = 0) &= v_1(0) \end{aligned}$$

4.3. Requirements

For this model we will specify a total of ten requirements, five of them probabilistic and five of them not. Since we will use three very different approaches to implement and verify the model, we will phrase the requirements using a combination of natural language and mathematical operators.

From this point onward, we will assume $v_{1,max} = v_{2,max} = 10$ and $e_{1,max} = 0.5, e_{2max} = 0.05$ unless explicitly stated otherwise.

- Non-probabilistic requirements:

  NP1 Does $r \geq (v_{1,\text{raw}} + v_{2,\text{raw}}) \cdot 1.1$ eventually hold true in at least one iteration?

  NP2 Does $(v_{1,\text{raw}} + v_{2,\text{raw}}) \cdot 0.9 - 0.5 \leq r \leq (v_{1,\text{raw}} + v_{2,\text{raw}}) \cdot 1.1 + 0.5$ eventually hold true in each and every iteration?

  NP3 Assuming that $e_2 \leq 0, v_{1,\text{raw}} = 0$ holds, does $r > v_{2,\text{raw}}$ eventually hold true in at least one iteration?

  NP4 Assuming that $v_{1,\text{raw}} \geq 1$, does $r_p > 10 \cdot v_{1,\text{raw}}$ eventually hold true in at least one iteration?

  NP5 Does $r_p \geq v_{1,\text{raw}} + v_1 + v_2$ eventually hold true in at least one iteration?

- Probabilistic requirements:

  P1 What is the probability that $r_w \geq 10$ eventually holds true when $w \geq 0.7$ holds?

  P2 What is the probability that $r_w \geq 10$ eventually holds true when $(v_{1,\text{raw}} + v_{2,\text{raw}}) \geq 10$ holds?

  P3 What is the probability that $r_w > 20$ eventually holds true?

  P4 Does the probability of $r \geq (v_{1,\text{raw}} + v_{2,\text{raw}})$ equal that of $r \leq (v_{1,\text{raw}} + v_{2,\text{raw}})$?

  P5 What is the probability that $r_p \geq 2 \cdot v_{1,\text{raw}}$ eventually holds true?

The requirements can be considered to be rather complex and mainly depend on the ‘raw’ input parameters such that the error models are incorporated for each parameter.

5. Evaluation

We are comparing different tools with varying strengths to implement the model and verify the requirements. The used languages and tools have been introduced in Section 2.3 and the model building and verification process is documented in this chapter.

5.1. Setup

The calculations have been performed inside a VMWare appliance with 4 vCPU's, each detected as an Intel E5-2683 v4 @2.10 GHz with two cores and a total BogoMIPS of 4200 and 64GB of RAM. The applications used for this evaluation run mostly single-threaded, with some of the calculations – if possible – offloaded to other threads.

The Storm model checker was compiled from source version 1.3.0 with the optional Intel Threaded Building Blocks, MathSAT and Gurobi libraries enabled. For the PRISM model checker, Lustre & Kind 2 as well as mCRL2 we used the pre-compiled binaries.

5.2. Synchronous Dataflow Programming Language: Lustre & Kind 2

In this section we will implement the model from Section 4.2 in Lustre, the requirements from Section 4.3 as a Kind 2 contract.

5.2.1. Model Building

For our model, we will do the computations in a node where the raw parameters are input variables, the 'measured' variables are inner variables and the output is declared as node output. The computation node including the line specifying the corresponding contract is shown in Listing 5.1.

```
node Model(v1raw, v1e, v2raw, v2e : real) returns (r, rp : real);
(*@contract import ModelSpec(v1raw, v1e, v2raw, v2e) returns (r, rp); *)
var
    v1, v2 : real;
let
    v1 = v1raw + v1e;
    v2 = v2raw + v2raw * v2e;

    r = v1 + v2;
    rp = v1 + 0.0 -> pre(v2);
tel
```

Listing 5.1.: Lustre node to calculate r, r_p

Implementing the model behaviour in Lustre is straightforward, since all properties except stochastic input parameters are natively supported and Lustre can work with floating point numbers (called `Real` in Lustre) without any need for discretization.

5.2.2. Verification

Since KIND 2 only verifies safety properties, we will need to invert reachability properties. To give an example, instead of checking whether `r = 21` can be reached, we attempt to check the inverse: guaranteeing that it cannot be reached and seeing whether the model checker outputs `false` as well as a trace on how this is obtained. We have transferred the requirements from Section 4.3 to a KIND 2 contract in Listing 5.2. The contract starts with several `assumes`, limiting the range of the input parameters and ensuring that v_1, v_2 never get negative. Then, a mode `def` is defined, which is the default mode and always holds true; when using mode-based verification, at least one mode must be matched, so we include this to guarantee a successful verification. The requirements NP1, NP2 and NP5 could be expressed using `guarantee` statements since the dependence on the input parameters can be directly encoded into the query. For the requirements NP3 and NP4 we had to specify modes to indicate for which range of the input parameters the guarantee should hold true.

```
contract ModelSpec(v1raw, v1e, v2raw, v2e : real) returns (r, rp : real);
let
    -- limits for the raw input parameters
    assume v1raw >= 0.0;
    assume v1raw <= 10.0;
    assume v2raw >= 0.0;
    assume v2raw <= 10.0;
    -- limits for the errors
    assume v1e >= -0.5;
    assume v1e <= 0.5;
    assume v2e >= -0.05;
    assume v2e <= 0.05;
    -- ensure that the calculated values stay within defined bounds (v1, v2 not get
    ↪  negative)
    assume (v1raw + v1e) >= 0.0;
    assume (v1raw + v1e) <= 10.5;
    assume (v2raw + v2raw * v2e) >= 0.0;
    assume (v2raw + v2raw * v2e) <= 10.5;

    -- default mode, required to perform mode-based model-checking
    mode def (
        require v1raw >= 0.0;
        require v2raw >= 0.0;
    );

    guarantee not (r >= (v1raw + v2raw) * 1.1); -- NP1
    guarantee r <= ((v1raw + v2raw) * 1.1 + 0.5) and (r >= (v1raw + v2raw) * 0.9 -
    ↪  0.5); -- NP2

    mode NP3 (
        require v2e <= 0.0;
        require v1raw = 0.0;
        ensure not (r > v2raw);
    );

    mode NP4 (
        require v1raw >= 1.0;

        ensure not (rp > 10.0 * v1raw);
    );

    guarantee not (rp >= v1raw + v1 + v2); -- NP5
tel
```

LISTING 5.2.: Lustre contract for node P

Note that every requirement except NP2 expresses a requirement does not need to always hold true, which in LTL-like specification languages would be denoted using the *exists* operator. Since Lustre only verifies guarantees/ensures, we have to invert the property to express that it will never be true and if the model-checker gives us a counter-example, we know that the requirement holds true in at least one system state. The query and verification results are listed in Table 5.1; if the query had to be inverted to check for the property, the verification result is the opposite of the query result.

Req.	Query	Tool Output	Verification Result
NP1	`guarantee not (r >= (v1raw + v2raw) * 1.1)`	`property invalid`	✓
NP2	`guarantee r <= ((v1raw + v2raw) * 1.1 + 0.5) and (r >= (v1raw + v2raw) * 0.9 - 0.5)`	`property valid`	✓
NP3	`mode NP3`	`property invalid`	✓
NP4	`mode NP4`	`property invalid`	✓
NP5	`guarantee not (rp >= v1raw + v1 + v2)`	`circular dependency`	ϟ

Legend: ✓ Satisified, ✖ Violated, ϟ Not verifiable

TABLE 5.1.: Verification of the Lustre model using KIND 2

The properties defined for the requirements NP1, NP3 and NP4 were found to be invalid by the KIND 2 model checker and counter-examples were returned. Since these three properties describe requirements that *eventually hold true in at least one iteration*, we inverted these to guarantee that no state is found in which they are true. As the model checker found counter-examples, we know that the requirements are indeed valid in at least one iteration and that the requirements hold.

NP2 describes a requirement that has to be fulfilled *in each and every iteration*, which is why we do not have to invert the logic for the `guarantee` statement. The model checker found this property to be valid, such that we know that this requirement is satisfied as well.

Unfortunately it was not possible to verify NP5 here, since Kind 2 throws a *circular dependency* error. To resolve the circular dependency, both `v1` and `v2` would have to be removed from the `guarantee` statement since `rp` is directly calculated from them (or their value in the previous iteration), making the verification of this query not possible using this toolchain.

5.3. Process Algebrae: mCRL2

The next language we will implement the model in is mCRL2. mCRL2 describes mainly the behaviour of processes, actual data is not considered. There is no support for any kind of memory concept or the `pre`-operator, since no iteration-based concept could be implemented and thus no data can be preserved over iterations. Unfortunately, mCRL2 has no support of real numbers in the `sum`-operator for enumeration such that manual discretization is required. Attempting to use the built-in distribution operators disables verification, effectively making the built-in probabilistic operations useless for us since there is no 'easy' way to verify stochastic processes.

Because of these reasons, our mCRL2 model is the smallest, since we only realized the verification of r. Still, we managed to implement the error models as well, which might prove useful for the verification of other, different reactive systems.

5.3.1. Model Building

For our model, we declare the output functions as parametrizable actions and will get input from process variables enumerated by a bounded `sum`-operator. For this, we will need bounded variables from an enumerable domain. The variables `v1raw, v2raw` are limited to the value interval between 0 and $v_{1,\max}$ and $v_{2,\max}$ respectively, the same is done for `e1` and `e2` with respect to $e_{1,\max}$, $e_{2,\max}$. Then, before doing the actual computation, we also set the bounds for the computed parameters `v1` and `v2`. Note that all real numbers are given as fractions, since mCRL2 does not allow writing them any other way. The verification is done by checking the reachability of parametrized actions in the generated LTS, in this case of r.

```
% parametrized action to check for the calculation result
act r: Real;

proc P = sum v1raw, v2raw: Nat, v1e, v2e: Int . % enumerate raw, error input
 ↪  parameters
         % bounds for input parameters
         ((0 <= v1raw && v1raw <= 10) && (0 <= v2raw && v2raw <= 10) &&
          (v1e >= -5 && v1e <= 5) && (v2e >= -5 && v2e <= 5) &&
         % ensure that calculated values stay within bounds
          (((v1raw + (v1e/10)) >= 0) && ((v1raw/10 + (v1e/10)) <= 105/10)) &&
          (((v2raw + (v2raw * (v2e/100))) >= 0) && ((v2raw + (v2raw * (v2e/100))) <=
           ↪  105/10)))
         % calculate r and use parametrized action to traverse, restart process
```

```
      -> r((v1raw + (v1e/10)) + v2raw + (v2raw * (v2e/100))) . P;

init P;
```

LISTING 5.3.: MCRL2 code to calculate the model

The generated LTS from Listing 5.3 is nondeterministic with only one state, 1891 action labels and 14036 transitions. Unfortunately, while Listing 5.3 calculates `r` depending on the input parameters, we have no way to access the values of the input parameters from verification queries. For this, we add additional actions that transition using the chosen value as parameter, such that we can effectively determine which parameters led to the current state; this is shown in Listing 5.4.

```
% parametrized actions for the input values
act v1raw, v2raw: Nat;
act v1e, v2e: Int;
% parametrized actions for the calculated values
act v1, v2, r: Real;

proc P = sum v1r, v2r: Nat, v1err, v2err: Int . % enumerate raw, error input
 ↪  parameters
         % bounds for input parameters
         ((0 <= v1r && v1r <= 10) && (0 <= v2r && v2r <= 10) &&
          (v1err >= -5 && v1err <= 5) && (v2err >= -5 && v2err <= 5) &&
         % ensure that calculated values stay within bounds
          (((v1r + (v1err/10)) >= 0) && ((v1r/10 + (v1err/10)) <= 105/10)) &&
          (((v2r + (v2r * (v2err/100))) >= 0) && ((v2r + (v2r * (v2err/100))) <=
           ↪  105/10)))
         % traverse states using parametrized actions to allow for model checking
      -> v1raw(v1r) . v2raw(v2r) . v1e(v1err) . v2e(v2err)
         % calculate values for v1, v2
       . v1(v1r + (v1err/10)) . v2(v2r * (v2err/100))
         % calculate value for r and restart process
       . r((v1r + (v1err/10)) + v2r + (v2r * (v2err/100))) . P;

init P;
```

LISTING 5.4.: MCRL2 code to calculate model and enter a chain of states allowing to verify properties depending on input parameters

The LTS generated by the specification from Listing 5.4 is nondeterministic as well, with 84217 states, 2098 action labels and 98252 transitions.

5.3.2. Verification

To verify the requirements, we will have to express them as MCRL2-specific textual representations of μ-calculus formulae. Since we have only modeled the calculation of r in MCRL2, we are only able to verify the requirements NP1 – NP3 here.

For our data-dependent purposes, the use of both the modeling and verification language of MCRL2 turned out to be rather cumbersome, but the μ-calculus is very powerful and the imple-

Req.	Query	Tool Output	Verification Result

NP1	`nu X.([`
	`     exists v1r: Nat.v1raw(v1r)`
	`  => exists v2r: Nat.v2raw(v2r)`
	`  => exists x: Real.r(x)`
	`  => val(x>(11/10*(v1r+v2r)))`
	`    ]true)`
		`true`	✓

NP2	`nu X.([`
	`     forall v1r: Nat.v1raw(v1r)`
	`  => forall v2r: Nat.v2raw(v2r)`
	`  => forall x: Real.r(x)`
	`  => val(`
	`       (v1r + v2r) * (9/10) - 5/10 <= x ||`
	`       (v1r + v2r) * (11/10) + 5/10 >= x)`
	`    ]true)`
		`true`	✓

NP3	`nu X.([`
	`     v1raw(0)`
	`  => exists v2r: Nat.v2raw(v2r)`
	`  => exists e2: Nat.v2e(e2)`
	`  => val(e2 <= 0)`
	`  => exists x: Real.r(x)`
	`  => val(x > v2r)`
	`    ]true)`
		`true`	✓

Legend: ✓ Satisified, ✖ Violated, ϟ Not verifiable

TABLE 5.2.: Verification of the MCRL2 model of our case study

mentation here allows us to check whether states are reached using a combination of mathematical quantifiers and the implication operator =>.

In each query we use the greatest fixed point operator `nu` to search for a sequence of states in which the parametrized actions are traversed in a way that match the original requirement. We can parametrize the states using the `exists` and `forall` operators, which work like the known quantifiers, to later use them in the `val` function which takes an equation and outputs `true` if it holds, `false` otherwise.

For NP1 and NP3, we use the `exists` operator to check whether there is a sequence of states that matches the requirement, for NP2 we use the `forall` quantifier requiring the relation to hold for every valid sequence.

With the model from Listing 5.4 we could successfully check the requirements NP1 – NP3 and the results are consistent with the ones obtained using Lustre and the KIND 2 model checker.

5.4. Probabilistic Model-Checkers: PRISM, Storm

This approach using probabilistic model checkers differs a bit from the other two approaches since the used tools are not explicitly designed to verify reactive systems. But since PRISM and Storm natively support stochastic verification, which the others do not, and is very flexible in what models can be created, we will attempt to use it to our purpose. The PRISM model can also be used as input for the Storm model checker[11], which implements comparable functions using different engines.

As mentioned in Section 4.1 we will have to choose a discretization for the probability distribution, which will have an impact on the verification outcome. We will choose a step size res_w such that w is evaluated $\frac{1}{\text{res}_w}$ times between $[a, b]$; for example, with $\text{res}_w = 0.1$ we would have 10 steps from 0.1 to 1.0 while with $\text{res}_w = 0.01$ we would have 100 steps from 0.01 to 1.0.

5.4.1. Model Building

We will build our model as an MDP and will start by introducing global variables (the 'raw' input parameters to be initialized) as well as variables that indicate whether the initialization has already taken place. In addition to the original model, we introduce a constant to limit the number of iterations in the model. Since the introduction of the history, unbounded iterations simply time out and it turns out that the state space grows exponentially with the amount of iterations chosen.

```
mdp

// raw values 0 to 10
global v1raw: [0..10];
global v2raw: [0..10];
// error -0.5 to 0.5
global v1e: [-5..5];
// error -0.05 to 0.05
global v2e: [-5..5];
// weighting 0 to 1
global w: [0..100];

// boolean initialization variables
global v1rawInit: [0..1] init 0;
global v2rawInit: [0..1] init 0;
global v1eInit: [0..1] init 0;
global v2eInit: [0..1] init 0;
global wInit: [0..1] init 0;

// number of iterations
const int k = 1;
global i: [0..k];
```

Listing 5.5.: Header for the PRISM model, defining global variables

The variable `v1` is initialized in the module `initv1raw`, which nondeterministically assigns a variable inside the bounds and sets `v1rawInit` to 1 (equivalent to true) meaning it is initialized.

In our example, v1 and v2 share the same bounds, so we can simply use a module rewrite of this module for the initialization of v2.

```
module initv1raw
    [] (v1rawInit = 0) -> (v1raw' = 0)  & (v1rawInit' = 1);
    [] (v1rawInit = 0) -> (v1raw' = 1)  & (v1rawInit' = 1);
    ...
    [] (v1rawInit = 0) -> (v1raw' = 10) & (v1rawInit' = 1);
endmodule

module initv2raw = initv1raw [v1raw = v2raw, v1rawInit = v2rawInit] endmodule
```

Listing 5.6.: Initialization of v_1, v_2 in PRISM

We can rewrite this MDP as an DTMC using uniformly distributed probabilities, which cuts down on the number of choices but does not impact the number of states or transitions. Such a rewrite is shown in the module initv1rawDTMC.

```
module initv1rawDTMC
    [] (v1rawInit = 0) -> 1/11 : (v1raw' = 0)  & (v1rawInit' = 1) +
                          1/11 : (v1raw' = 1)  & (v1rawInit' = 1);
                          ...
                          1/11 : (v1raw' = 10) & (v1rawInit' = 1);
endmodule
```

Listing 5.7.: Initialization of v_1 as DTMC in PRISM

Analogous to the initialization in initv1raw we can proceed with the error models. We will initialize these starting from a negative value and since both v1e and v2e share the same bounds, we can again use a rewrite to initialize the latter easily.

```
module initv1e
    [] (v1eInit = 0) -> (v1e' = -5) & (v1eInit' = 1);
    [] (v1eInit = 0) -> (v1e' = -4) & (v1eInit' = 1);
    [] (v1eInit = 0) -> (v1e' = -3) & (v1eInit' = 1);
    ...
    [] (v1eInit = 0) -> (v1e' = 5)  & (v1eInit' = 1);
endmodule

module initv2e = initv1e [v1e = v2e, v1eInit = v2eInit] endmodule
```

Listing 5.8.: Initialization of e_1, e_2 in PRISM

Initializing the weighting takes a bit more work. We will need to transfer the PDF to the PRISM model, which we have opted to do using a Python script. After choosing res_w we integrate the PDF from 0 to 1 in $\frac{1}{\text{res}_w}$ steps of res_w and set the w to the respective step times $\frac{1}{\text{res}_w}$. Using this method, we need to either omit the very first value (0) or the very last (1); here, we opted to let the sequence run from 1 to 100, such that w here could never actually reach 0, although in the actual model it could with a very low probability.

Note that res_w has a notable impact on the total model complexity. It not only determines the bounds of `w`, but also the complexity of the initialization routine and the bounds of parameters calculated using `w`.

```
module initw
    [] (wInit = 0) -> 0.00720808394646841 : (w' =   1) & (wInit' = 1) +
                      0.00729191001115681 : (w' =   2) & (wInit' = 1) +
                      0.0073755584209887  : (w' =   3) & (wInit' = 1) +
                      ...
                      0.0106112725180697  : (w' = 100) & (wInit' = 1);
endmodule
```

Listing 5.9.: Initialization of w in PRISM

We have tested several values for res_w while not modifying any of the other parameters and got the results listed in Table 5.3 for the amount of states and transitions.

res_w	0.2	0.1	0.05	0.02	0.01
States	247988	757320	1705464	2723428	7792136
Transitions	648673	1679322	3635190	6377309	16733386

Table 5.3.: Effect of different values of res_w on state and transition amount

After all the initialization is done, we will do the actual calculations and transformations. In the module `simple` we initialize the local variables with the correct bounds and advance the global `state` variable to 1 as soon as all 'raw' variables are initialized. Then we use two disjunct calculation routines for `v1` and `v2` to ensure that `v1` does not get below 0.

Since we must use integers for the calculations and cannot divide here, the bounds of the calculated variables are way larger than that of their initialized counterparts. After `v1` and `v2` are initialized, we first initialize `r` and then `rw` and `rp`, also saving the current value of `v2` in `prev2` to keep the history. If the iteration count specified by `k` is not yet met, we reset both the initialization variables as well as `state` to proceed with the next iteration; note that each iteration exponentially increases complexity.

```
module model
       // computed values in range 0 to 10.5
       v1 : [0..1050];
       v2 : [0..1050];
       // computed r in range 0 to 21
       r : [0..2100];
       // computed rw in range 0 to 21
       rw : [0..210000];
       // value v2 from previous iteration in range 0 to 10.5
       prev2 : [0..1050] init 0;
       // computed rp in range 0 to 21
       rp : [0..2100];
```

```
        // state variable to ensure sequential execution order
        state: [0..5] init 0;

        // begin calculations when everything is initialized
        [] (v1rawInit = 1 & v2rawInit = 1 & v1eInit = 1 & v2eInit = 1 & wInit = 1 &
        ↪  state = 0) -> (state' = 1);

        // calculate v1, v2 and ensure that v1 never gets negative
        [] (state = 1 & (v1e >= 0 | 10 * v1raw > -1 * v1e)) -> (v1' = v1raw * 100 + v1e
        ↪  * 10) & (v2' = v2raw * 100 + v2raw * v2e) & (state' = 2);
        [] (state = 1 & (v1e < 0 & 10 * v1raw <= -1 * v1e)) -> (v1' = v1raw * 100) &
        ↪  (v2' = v2raw * 100 + v2raw * v2e) & (state' = 2);

        // calculate r after v1, v2
        [] (state = 2) -> (r' = v1 + v2) & (state' = 3);

        // calculate rw, rp and save v2 for next iteration
        [] (state = 3) -> (rw' = r * w) & (rp' = v1 + prev2) & (prev2' = v2) & (state'
        ↪  = 4);

        // perform k iterations, then transition to final state, ending calculations
        [] (state = 4 & i < (k - 1)) -> (v1rawInit' = 0) & (v2rawInit' = 0) & (v1eInit'
        ↪  = 0) & (v2eInit' = 0) & (wInit' = 0) & (state' = 0) & (i' = i + 1);
        [] (state = 4 & i >= (k - 1)) -> (state' = 5);
endmodule
```

Listing 5.10.: Main module of PRISM model, handling the actual calculations of r, r_p, r_w

5.4.2. Verification

In PRISM, we can verify all given requirements, both probabilistic and non-probabilistic. We will express these as properties and verify them using the Storm model checker, which can take PRISM models as input but in our tests turned out to be a lot faster.

Note that since we have created an MDP model, we cannot simply check for probabilities using `P=?`, we have to check for the minimum or maximum probability given the nondeterministic choices, which is done using the `Pmin`, `Pmax` operators respectively.

Note that in the queries listed in Table 5.4, `&` denotes a logical AND, `|` denotes a logical OR and `||` denotes a dependence, such that e.g. `P=? [ F b>10 || F a<10 ]` returns the probability of `b>10` eventually being true given that `a<10` holds. We use this to make all queries depend on `state = 5`, signaling a finished calculation.

The requirements NP1 – NP3 could be checked just like in the previous two tools and all probabilities returned are equal to 1, which means that the requirements are met. Note that for NP1 and NP3, we use `Pmax` to check for the maximum probability of the requirement being met – since it does only need to hold in at least one iteration – but for NP2 we use `Pmin` to check for the minimum probability as it needs to always hold.

Unfortunately, NP4, NP5 and P5 – every query requiring `k = 2` – makes the Storm model

Req.	Query	Tool Output	Verification Result
NP1	`Pmax=? [ F r >= 110 * (v1raw + v2raw) || F state = 5 ]`	1	✓
NP2	`Pmin=? [ state != 5 U (state = 5 & (r <= 110 * (v1raw + v2raw) | r >= 90 * (v1raw + v2raw))) ]`	1	✓
NP3	`Pmax=? [ F r >= v2raw || F state = 5 & (v1raw = 0 & v2e <= 0) ]`	1	✓
NP4	`Pmax=? [ F rp > 1000 * v1raw || F state = 5 ]`	OOM	⚡
NP5	`Pmax=? [ F rp >= 100 * v1raw + v1 + v2 || F state = 5 ]`	OOM	⚡

P1	`Pmin=? [ F rw >= 100000 || F (state = 5 & w >= 70) ]`
	`Pmax=? [ F rw >= 100000 || F (state = 5 & w >= 70) ]`
		0
		1	

P2	`Pmin=? [ F rw>=100000 || F (state=5 & (v1raw+v2raw)>=10) ]`
	`Pmax=? [ F rw>=100000 || F (state=5 & (v1raw+v2raw)>=10) ]`
		0
		1	

P3	`Pmin=? [ F rw > 200000 || state = 5 ]`
	`Pmax=? [ F rw > 200000 || state = 5 ]`
		0
		0.0534	

P4	`Pmin=? [ F r >= 100 * (v1raw + v2raw) || F state = 5 ]`
	`Pmin=? [ F r <= 100 * (v1raw + v2raw) || F state = 5 ]`
	`Pmax=? [ F r >= 100 * (v1raw + v2raw) || F state = 5 ]`
	`Pmax=? [ F r <= 100 * (v1raw + v2raw) || F state = 5 ]`
		0
		0
		1
		1	

P5	`Pmin=? [ F rp >= 200 * v1raw || F state = 5 ]`	OOM	⚡

Legend: ✓ Satisified, ✖ Violated, ⚡ Not verifiable

TABLE 5.4.: Verification of the PRISM model of our case study

checker eventually run out of memory during the model building step, making them unverifiable using our setup. We are able to check simpler requirements using the `exploration` engine of STORM for `k` = 2, since it does not build the full model but only the subset required for the verification. But since the `exploration` engine does not support the dependency operator, we cannot use it for our queries here. Reducing the resolution of the weighting parameter also does not help to build the model, the state space needs to be reduced drastically for the model to build requiring a change of $v_{1,max}, v_{2,max}, e_{1,max}$ and e_{2max}. Since this would change the model behaviour, we consider these queries to be unverifiable here; how the parameters affect the state space and for which parameters a model could be built is detailed in Section 6.2.

For the probabilistic requirements P1 – P4 we get minimum and maximum probabilities as well, but without knowing the distribution for the errors, we cannot reliably give a probability estimate for these requirements; we only know that they hold in some cases and do not in others. Note that for P3, we get a very small maximum probability. Looking at the query, the reason is obvious: for $r_w > 20$, all input parameters need to be quite large and very near their upper bound and the weighting needs to be very close to 1. The maximum probability for P3 is thus the probability that, given $v_1 = v_2 = 10, e_1 = 0.5, e_2 = 0.05$, w is large enough such that $r_w > 20$ holds true.

5.5. Summary

Having implemented the model, or at least parts of it, in all languages, we can now summarize the results.

	mCRL2	Lustre + Kind 2	PRISM / Storm
floating point numbers	📄	📄 ⚙	📄
error models	</> ⚙	</> ⚙	</> ⚙
stochastic input	📄		📄 ⚙
access to data from previous iterations		📄 ⚙	</> ⚙
unlimited iterations	📄 ⚙	📄 ⚙	📄 ⚡

Legend: 📄 Native support, </> Implementable, ⚙ Verifiable, ⚡ Problems during verification

Table 5.5.: Summary of the tool applicability after the evaluation

For every tool but Lustre & Kind 2, floating point numbers are supported, but not in verification. For the verification, values from enumerable domains need to be used, such that discretization is required.

With the parts of the model implemented in mCRL2, unlimited iterations (by eventually ending up in the initial state again) are possible. Since Lustre uses an iteration-based concept natively, it can deal with checking an unlimited amount of iterations, including support for using data from an arbitrary amount of previous iterations.

For the PRISM model, we were able to implement every single property of the model, but at the cost of a very large and exponentially growing state space. With the model parameters used in this chapter, no more than three iterations could be verified and no full model could be build for more than two iterations. Attempting to verify more iterations resulted in an out-of-memory error.

6. Discussion

After having implemented the mathematical model in the three different languages and having performed the verification, we can discuss the results obtained and the challenges faced.

6.1. Tool Applicability

As mentioned in Section 1.3, Lustre and Kind 2 have been successfully used for the verification of SIMULINK models and other safety-critical reactive systems. It would have been a great match for our problem if it were not for the uncertainty in the input parameters, as implementing the basic calculations and those involving the history was straightforward. Only a single non-probabilistic requirement could not be verified due to a circular dependency. For some parts of the function without such uncertainty, we can use Lustre and Kind 2 to verify them as a closed system and use the results to reason about the system as a whole.

Although explicitly designed for reactive systems, mCRL2 was the least suited tool for our model. Its strengths lie in the specification and verification of highly parallel communicating systems, but it was not well-suited for our very data-intensive, complex but rather sequential model. The enumeration of possible input parameters required discretization that severely limited the scope of the implementation, an implementation of the history-dependent r_p was not possible since dynamic data could not be saved or read and the verification of parameters depending on enumerated input parameters was rather cumbersome. The tool also advertises support for probabilistic inputs and giving input distributions for input parameters is possible, but with such distributions specified, verification is not yet possible, only simulation, which leads to the feature not being applicable in our case.

Using the PRISM specification, we were able to implement the complete model including the stochastic parameter, although we had to apply discretization to the probability distribution function. Unlike with the other models, the complexity in the PRISM model grew so high that we were required to severely limit the amount of iterations and even then we were not able to verify all requirements without further reducing the complexity by scaling down the maximum values of the parameters.

To summarize, even though we designed a seemingly simple mathematical model, it provided considerable challenges for existing model-checking approaches. We have used state-of-the-art tools to attempt to verify the requirements on a powerful machine and were not able to obtain reliable results for every part of the system in a single tool.

6.2. Complexity Estimation

Being able to estimate the complexity of a model is very valuable, especially when transitioning from academic use to industry use, where projects are oftentimes orders of magnitude larger than in academia, in which models are often closer to small proof-of-concepts.

In our model, we dynamically changed the following bounds and recorded both the number of states and transitions (in case the model could be built) to estimate the complexity: $v_{1,max}$, $v_{2,max}$, $e_{1,max}$, e_{2max} as well as res_w. For the PRISM model, the change in parameters led to the changes displayed in Table 6.1.

$v_{1,\text{max}}$	$v_{2,\text{max}}$	$e_{1,\text{max}}$	$e_{2,\text{max}}$	res_w	k	states	transitions	time (s)	mem (Byte)
1	1	1	1	5	1	1331	3955	23	49532
3	3	2	2	10	1	27780	64400	128	52308
5	5	3	3	20	1	406896	886004	1645	102888
10	10	5	5	50	1	2723428	6377309	10544	334060
10	10	5	5	100	1	7792136	16733386	30744	944532
5	5	1	1	20	2	289530187	735988121	1241919	31280316
5	5	3	3	20	2		⚡		
10	10	5	5	100	2		⚡		

TABLE 6.1.: Effect of the input parameter range on the model complexity

This is just an excerpt of our measurements, but it shows how even small changes to the range of input parameters have increasingly large effects on both the complexity of the generated model and the time and memory required to build it.

For the last two listed parameter configurations, a full model build was not possible, but verification results could be attained using the `exploration` engine of the STORM model checker for supported properties.

We can also rewrite the MDP as a DTMC by assigning equal probabilities to all nondeterministic transitions. Representing the MDP as an DTMC, all choices are eliminated and the model becomes fully deterministic. This does severely affect the verification results, though, since choosing the input parameters now affects the probability of the output; if the distribution is not known, this change eliminates the model validity.

In a similar fashion, we can change the header of the DTMC-converted model to represent an MDP. When we do this to indicate an MDP to the model checker which actually is a DTMC, choices are re-introduced, but model building time using STORM is almost not affected. As Table 6.2 shows, neither of those actions affect the amount of states or transitions.

	DTMC	MDP	DTMC as MDP
States	7792136	7792136	7792136
Transitions	16733386	16733386	16733386
Choices		14721994	7879114
Time to build	30s	42s	31s

TABLE 6.2.: Effect of changing the model type on state, transition and choice amount

For our verification, we have used the MDP model, which – according to the time to build and the amount of choices – is the most complex one. Selecting any of the other leads to faster model building and verification times, overall lower complexity – due to less or no nondeterministic choices – but affects the output probability. If an MDP turns out to not be verifiable, it might be worth using this method to attempt verification again, but the effect on output probability needs to be thoroughly calculated beforehand and considered during verification.

6.3. Threats to Validity

Even given this small model, we were not able to verify an unmodified version of it in any of the chosen tools. We had to work with discretization everywhere, which is to be expected when implementing a model in any kind of computer language, but even this is already a stray from the purely mathematical concept of the model.

Additionally, each language and tool had its own shortcomings that had to be worked with. Both Lustre and mCRL2 do not support stochastic verification, so part of the model could not be implemented. In the PRISM model, we were able to implement every part of the model, but had to severely limit the amount of iterations making it more of a simulation of the model than an actual implementation.

Although mCRL2 has built-in verification support for floating point numbers, verification of inputs in a given range requires enumeration using the `sum` operator, which does not support the `Real` data type as it is not enumerable. This means that we had to enumerate the inputs using the domain of natural numbers and such apply large-style discretization, limiting the validity of the resulting verification results.

The Lustre model itself can work with floating point numbers without problems, such that the Lustre model is the only one not requiring any discretization whatsoever. During verification, though, the Kind 2 solver emits several warnings which need to be kept in mind when interpreting the verification results.

The amount of discretization required to implement the model in mCRL2 drastically limits the validity of the results. Since only enumerable domains could be worked with, choosing $v_{1,\text{raw}}, v_{2,\text{raw}}, e_1$ and e_2 from natural numbers only made the set of possible outcomes of this implementation very small compared to the others.

Since the PRISM model allows for stochastic verification but does not natively support the probability distribution function, the amount of discretization required for this model is the highest, since the PDF needs to be converted to a module with probabilities assigning values to w. In the evaluation we chose a high enough discretization to achieve pretty reliable results, but this also resulted in an increased complexity of the overall model, aggravating the verification of multiple iterations.

7. Conclusion

We have developed a small yet complex mathematical model and attempted to implement it and verify requirements imposed over it using three different model-checking approaches for reactive systems. During the implementation of the model in the different languages, we have found and partly solved challenges unique to the language-specific modeling approach. With the state-of-the-art tools available, we did not succeed in fully verifiying all requirements with a single tool.

This highlights the sheer complexity of model-checking modern systems, where properties like error models and unreliable input need to be taken into consideration. The complexity of model-checking systems has been a problem for decades and while several approaches have been developed to attempt to deal with the exploding state space problem, vastly growing model complexity and size are to be dealt with.

Regarding the properties specified in the beginning, we can now estimate how they affect the complexity of the model and how well state-of-the-art tools can handle them. In the end, none of the surveyed tools was able to completely verify our model to the full extent given the requirements we specified.

Support for floating point numbers is present in all surveyed tools to some extent, although only Kind 2 is able to do model-checking using the built-in floating point data type of Lustre, all other tools required some amount of discretization to emulate floating point numbers using the domain of natural numbers. As expected, the implementation of the error models was quite straight-forward in all tools then, since that essentially just required implementing ranges for the input parameters.

Implementing the history-dependent verification unfortunately was not possible in mCRL2, since it focuses on modeling the control flow of a system and does not provide support for saving and reading data in an iteration-based model. Lustre has built-in support for the `pre`-operator, which is a native implementation of the history concept; it can also be chained to go back multiple iterations. Kind 2 also supports verification of this operator, making the implementation of this part of the model very easy for these tools. For PRISM, no native `pre`-like operator is available, but we managed to implement the concept using an additional variable and an iteration-like state reset concept. Unfortunately using this approach, every additional iteration leads to a large exponential growth as can be seen in Section 6.2; with an increase of the iteration count k by one leading to an increase in states to $25 \cdot \#(k-1)^k$, where $\#(k)$ gives the amount of states for k iterations. Although a lot of techniques have already been applied by the model checker to reduce state space explosion, this does absolutely not scale well and for our model with the parameters used in Chapter 4 already fails to fully build for $k = 2$ iterations and does not allow for verification even using the `exploration`-engine for $k = 3$ iterations.

The unreliable input parameters led to the need for stochastic verification, which is not implementable in Lustre using only the very basic operators available. While mCRL2 does support specifying probability distributions for input parameters, the resulting models can only be simulated, not verified, such that in our case the feature was not applicable at all. Being probabilistic model checkers, PRISM and Storm natively support the verification of models with probabilistic input parameters. Although we had to discretize our probability distribution function for this approach, we were able to choose the discretization fine enough to achieve reliable results. Tool support for stochastic verification is not widely adopted as of yet such that – if this is a hard requirement for the choice of tools – the set of available options is rather small. Trade-offs have to be considered when opting for this approach, since the complexity of models of reactive systems turn out to be several scales larger for these tools than for 'traditional' model checking tools.

The concept of model checking is decades old and so is the development of various techniques to reduce the state space. Several of these methods will be introduced here, with references to literature with further information.

On-the-fly reduction[23] describes a group of techniques to reduce the state space during exploration, e.g. by recognizing duplicate/equivalent states and merging the paths, thus reducing the actual state space without losing any information in the model. There are various algorithms here that can be applied and most of them can be used together. One of the most well known is *partial order reduction*[8, 9, 23], which aims to reduce the possible orderings of asynchronous/parallel processes.

The category *predicate abstraction and refinement*[23] contains techniques to abstract states using both over- & under-approximation. An example of this is *counterexample-guided abstraction refinement*[8–10, 23] (CEGAR), which takes the property into account, over-approximates the model and abstracts away states that violate properties, while iteratively refining the over-approximated property to get closer to the actual verification query. For probabilistic systems, *abstraction-based refinement*[8] replaces probabilistic transitions with nondeterministic ones to abstract 'safe' states considering this over-approximation. *Counter-example guided refinements*[15] can also be applied to probabilistic models, working similar to CEGAR. *Under-approximation refinement*[23] carefully removes states in a less strict abstraction of the model while ensuring that all safety properties for the system are still valid. Taking the property into account during the verification process is also possible and used e.g. in an approach called *strategy synthesis* which has been shown to work with MDPs[21].

Bounded Model Checking[8, 9] (BMC) applies a technique similar to the one we used to limit the iterations in our PRISM model. Given a bound k, a formula is said to be valid if it cannot be disproven by a counterexample of length k. This effectively gives an upper bound for the generated state space, even if it otherwise were to be infinite, and is widely used in model checking tools available today. There are various methods that build on top of BMC, a lot of which also work with probabilistic systems[8].

Symbolic Model Checking[8, 9] (SMC) attempts to represent the model, given in a finite-state machine-like format, using boolean equations and using techniques to reduce and simplify the attained terms. Most of the approaches to SMC represent the formulae using binary decision diagrams, since there exist various algorithms to effectively generate minimized representations of them.

While reducing the model complexity is the most effective way to increase verification performance, it is not the only way towards a more efficient verification of large-scale models. We have developed our probabilistic model in the input format for the PRISM model checker and have extensively tested various configurations of it before we began also using the Storm model checker. Supporting the same input format, we were easily able to compare both model checkers. Being implemented in C++ instead of Java, the Storm model checker turned out to build and verify models faster by a scale of 10 while also achieving a lot lower RAM usage. We have found several parameter configurations for which the model was not verifiable using the PRISM model checker with which the Storm model checker successfully completed the verification.

While both PRISM and Storm have several verification engines available, only the Storm model checker provides the `exploration` engine, which is especially suited to models with a very large state space[11], due to this the usage of techniques using machine learning for the reduction of the state space[2]. Instead of building the full model and then performing verification, it takes the verification query into account from the very beginning and only builds the relevant parts of the model on-the-fly during verification, enabling it to check queries on models that could not be built fully due to a too large state space. Unfortunately, using the `exploration` engine, the dependency operator is not available for verification queries such that none of our requirements could be verified using this engine.

Our survey has shown that while there has been a lot of progress in model checking tools, with growing system complexity this problem is still a very challenging one to tackle. Even for a simplified case study of our original model, we were not able to attain fully reliable results using any of the used approaches, although we were able to verify several rather complex system properties, which of course is better than gaining no results or using test-based methods. With both further research into techniques to deal with state space explosion as well as even more mature, efficient implementations of model checking tools, we are sure that progress can be made to not only keep up with system and model complexity, but also to efficiently verify such models.

Bibliography

[1] Guillaume Brat et al. 'Verifying the Safety of a Flight-Critical System'. In: *FM 2015: Formal Methods*. Springer International Publishing, 2015, pp. 308–324. DOI: 10.1007/978-3-319-19249-9_20 (cit. on p. 3).

[2] Tomáš Brázdil et al. 'Verification of Markov Decision Processes Using Learning Algorithms'. In: *Automated Technology for Verification and Analysis*. Springer International Publishing, 2014, pp. 98–114. DOI: 10.1007/978-3-319-11936-6_8 (cit. on p. 35).

[3] David Broman et al. 'Viewpoints, formalisms, languages, and tools for cyber-physical systems'. In: *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling - MPM '12*. ACM Press, 2012. DOI: 10.1145/2508443.2508452 (cit. on p. 3).

[4] Olav Bunte et al. 'The mCRL2 Toolset for Analysing Concurrent Systems'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 21–39. ISBN: 978-3-030-17465-1 (cit. on p. 9).

[5] Radu Calinescu et al. 'Formal Verification With Confidence Intervals to Establish Quality of Service Properties of Software Systems'. In: *IEEE Transactions on Reliability* 65.1 (Mar. 2016), pp. 107–125. DOI: 10.1109/tr.2015.2452931 (cit. on p. 3).

[6] Adrien Champion et al. 'The Kind 2 Model Checker'. In: *Computer Aided Verification*. Springer International Publishing, 2016, pp. 510–517. DOI: 10.1007/978-3-319-41540-6_29 (cit. on p. 8).

[7] Jost-Pieter Katoen Christel Baier. *Principles of Model Checking*. The MIT Press, June 1, 2008. ISBN: 026202649X (cit. on pp. 5, 14).

[8] Edmund M. Clarke and Qinsi Wang. '2^5 Years of Model Checking'. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2015, pp. 26–40. DOI: 10.1007/978-3-662-46823-4_2 (cit. on p. 34).

[9] Edmund M. Clarke et al. 'Model Checking and the State Explosion Problem'. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1 (cit. on p. 34).

[10] Edmund Clarke et al. 'Counterexample-guided abstraction refinement for symbolic model checking'. In: *Journal of the ACM* 50.5 (Sept. 2003), pp. 752–794. DOI: 10.1145/876638.876643 (cit. on p. 34).

[11] Christian Dehnert et al. 'A Storm is Coming: A Modern Probabilistic Model Checker'. In: *Computer Aided Verification*. Springer International Publishing, 2017, pp. 592–600. DOI: 10.1007/978-3-319-63390-9_31 (cit. on pp. 24, 35).

[12] Michael Dierkes. 'Formal Analysis of a Triplex Sensor Voter in an Industrial Context'. In: *Formal Methods for Industrial Critical Systems*. Springer Berlin Heidelberg, 2011, pp. 102–116. DOI: 10.1007/978-3-642-24431-5_9 (cit. on p. 3).

[13] David Garlan. 'Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events'. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 1–24. DOI: 10.1007/978-3-540-39800-4_1 (cit. on p. 3).

[14] Xiaocheng Ge, Richard F. Paige, and John A. McDermid. ‘Analysing System Failure Behaviours with PRISM’. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*. IEEE, June 2010. DOI: `10.1109/ssiri-c.2010.32` (cit. on p. 3).

[15] Sergio Giro and Markus N. Rabe. ‘Verification of Partial-Information Probabilistic Systems Using Counterexample-Guided Refinements’. In: *Automated Technology for Verification and Analysis*. Springer Berlin Heidelberg, 2012, pp. 333 – 348. DOI: `10.1007/978-3-642-33386-6_26` (cit. on p. 34).

[16] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems (The MIT Press)*. The MIT Press, 2014. ISBN: 9780262027717 (cit. on p. 9).

[17] Falk Howar et al. ‘Rigorous Examination of Reactive Systems’. In: *Int. J. Softw. Tools Technol. Transf.* 16.5 (Oct. 2014), pp. 457 – 464. ISSN: 1433-2779. DOI: `10.1007/s10009-014-0337-y` (cit. on p. 3).

[18] Yi-Ling Hwong, Vincent J. J. Kusters, and Tim A. C. Willemse. ‘Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider’. In: *Fundamentals of Software Engineering*. Springer Berlin Heidelberg, 2012, pp. 174 – 189. DOI: `10.1007/978-3-642-29320-7_12` (cit. on p. 3).

[19] Norman L. Johnson, Samuel Kotz, and N. Balakrishnan. *Continuous Univariate Distributions, Vol. 1 (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 1994. ISBN: 9780471584957 (cit. on p. 7).

[20] M. Kwiatkowska, G. Norman, and D. Parker. ‘PRISM 4.0: Verification of Probabilistic Real-time Systems’. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV’11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585 – 591 (cit. on pp. 2, 3, 9).

[21] Marta Kwiatkowska and David Parker. ‘Automated Verification and Strategy Synthesis for Probabilistic Systems’. In: *Automated Technology for Verification and Analysis*. Springer International Publishing, 2013, pp. 5 – 22. DOI: `10.1007/978-3-319-02444-8_2` (cit. on p. 34).

[22] Gethin Norman and David Parker. *Quantitative Verification: Formal Guarantees for Timeliness, Reliability and Performance*. Tech. rep. The London Mathematical Society and the Smith Institute, 2014 (cit. on p. 3).

[23] Radek Pelánek. ‘Reduction and Abstraction Techniques for Model Checking’. Doctoral theses, Dissertations. Masaryk University, Faculty of Informatics, Brno, 2006 (cit. on p. 34).

[24] Daniela Remenska et al. ‘Using Model Checking to Analyze the System Behavior of the LHC Production Grid’. In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, May 2012. DOI: `10.1109/ccgrid.2012.90` (cit. on p. 3).

[25] Klaus Schneider. *Verification of Reactive Systems*. Springer Berlin Heidelberg, 2004. DOI: `10.1007/978-3-662-10778-2` (cit. on p. 3).

Appendix

This chapter contains the full code for all the models that have been used for the verification process. Further tools and documentation is available on the disk attached to this paper.

A. Lustre & Kind 2 Model

```
contract ModelSpec(v1raw, v1e, v2raw, v2e : real) returns (r, rp : real);
let
  -- limits for the raw input parameters
  assume v1raw >= 0.0;
  assume v1raw <= 10.0;
  assume v2raw >= 0.0;
  assume v2raw <= 10.0;
  -- limits for the errors
  assume v1e >= -0.5;
  assume v1e <= 0.5;
  assume v2e >= -0.05;
  assume v2e <= 0.05;
  -- ensure that the calculated values stay within defined bounds (v1, v2 not get
 ↪  negative)
  assume (v1raw + v1e) >= 0.0;
  assume (v1raw + v1e) <= 10.5;
  assume (v2raw + v2raw * v2e) >= 0.0;
  assume (v2raw + v2raw * v2e) <= 10.5;

  -- default mode, required to perform mode-based model-checking
  mode def (
    require v1raw >= 0.0;
    require v2raw >= 0.0;
  );

  guarantee not (r >= (v1raw + v2raw) * 1.1); -- NP1
  guarantee r <= ((v1raw + v2raw) * 1.1 + 0.5) and (r >= (v1raw + v2raw) * 0.9 - 0.5);
 ↪  -- NP2

  mode NP3 (
    require v2e <= 0.0;
    require v1raw = 0.0;

    ensure not (r > v2raw);
  );

  mode NP4 (
    require v1raw >= 1.0;

    ensure not (rp > 10.0 * v1raw);
  );

  guarantee not (rp >= v1raw + v1 + v2); -- NP5
tel

node Model(v1raw, v1e, v2raw, v2e : real) returns (r, rp : real);
(*@contract import ModelSpec(v1raw, v1e, v2raw, v2e) returns (r, rp); *)
var
```

```
  v1, v2 : real;
let
  v1 = v1raw + v1e;
  v2 = v2raw + v2raw * v2e;

  r = v1 + v2;
  rp = v1 + 0.0 -> pre(v2);
tel
```

B. mCRL2 Model

```
% parametrized actions for the input values
act v1raw, v2raw: Nat;
act v1e, v2e: Int;
% parametrized actions for the calculated values
act v1, v2, r: Real;

proc P = sum v1r, v2r: Nat, v1err, v2err: Int . % enumerate raw, error input parameters
         % bounds for input parameters
         ((0 <= v1r && v1r <= 10) && (0 <= v2r && v2r <= 10) &&
          (v1err >= -5 && v1err <= 5) && (v2err >= -5 && v2err <= 5) &&
         % ensure that calculated values stay within bounds
          (((v1r + (v1err/10)) >= 0) && ((v1r/10 + (v1err/10)) <= 105/10)) &&
          (((v2r + (v2r * (v2err/100))) >= 0) && ((v2r + (v2r * (v2err/100))) <=
          ↪  105/10)))
         % traverse states using parametrized actions to allow for model checking
      -> v1raw(v1r) . v2raw(v2r) . v1e(v1err) . v2e(v2err)
         % calculate values for v1, v2
       . v1(v1r + (v1err/10)) . v2(v2r * (v2err/100))
         % calculate value for r and restart process
       . r((v1r + (v1err/10)) + v2r + (v2r * (v2err/100))) . P;

init P;
```

C. PRISM Model

```
mdp

// raw values 0 to 10
global v1raw: [0..10];
global v2raw: [0..10];
// error -0.5 to 0.5
global v1e: [-5..5];
// error -0.05 to 0.05
global v2e: [-5..5];
// weighting 0 to 1
global w: [0..100];

// boolean initialization variables
global v1rawInit: [0..1] init 0;
global v2rawInit: [0..1] init 0;
global v1eInit: [0..1] init 0;
global v2eInit: [0..1] init 0;
global wInit: [0..1] init 0;

// number of iterations
const int k = 1;
global i: [0..k];

module model
        // computed values in range 0 to 10.5
        v1 : [0..1050];
        v2 : [0..1050];
        // computed r in range 0 to 21
        r : [0..2100];
        // computed rw in range 0 to 21
        rw : [0..210000];
        // value v2 from previous iteration in range 0 to 10.5
        prev2 : [0..1050] init 0;
        // computed rp in range 0 to 21
        rp : [0..2100];

        // state variable to ensure sequential execution order
        state: [0..5] init 0;

        // begin calculations when everything is initialized
        [] (v1rawInit = 1 & v2rawInit = 1 & v1eInit = 1 & v2eInit = 1 & wInit = 1 &
        ↪  state = 0) -> (state' = 1);

        // calculate v1, v2 and ensure that v1 never gets negative
        [] (state = 1 & (v1e >= 0 | 10 * v1raw > -1 * v1e)) -> (v1' = v1raw * 100 +
        ↪  v1e * 10) & (v2' = v2raw * 100 + v2raw * v2e) & (state' = 2);
        [] (state = 1 & (v1e < 0 & 10 * v1raw <= -1 * v1e)) -> (v1' = v1raw * 100) &
        ↪  (v2' = v2raw * 100 + v2raw * v2e) & (state' = 2);

        // calculate r after v1, v2
        [] (state = 2) -> (r' = v1 + v2) & (state' = 3);

        // calculate rw, rp and save v2 for next iteration
        [] (state = 3) -> (rw' = r * w) & (rp' = v1 + prev2) & (prev2' = v2) &
        ↪  (state' = 4);
```

```

            // perform k iterations, then transition to final state, ending calculations
            [] (state = 4 & i < (k - 1)) -> (v1rawInit' = 0) & (v2rawInit' = 0) &
            ↪  (v1eInit' = 0) & (v2eInit' = 0) & (wInit' = 0) & (state' = 0) & (i' = i +
            ↪  1);
            [] (state = 4 & i >= (k - 1)) -> (state' = 5);
endmodule

module initv1raw
        [] (v1rawInit = 0) -> (v1raw' = 0)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 1)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 2)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 3)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 4)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 5)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 6)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 7)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 8)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 9)  & (v1rawInit' = 1);
        [] (v1rawInit = 0) -> (v1raw' = 10) & (v1rawInit' = 1);
endmodule

module initv2raw = initv1raw [v1raw = v2raw, v1rawInit = v2rawInit] endmodule

module initv1e
        [] (v1eInit = 0) -> (v1e' = -5) & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = -4) & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = -3) & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = -2) & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = -1) & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 0)  & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 1)  & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 2)  & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 3)  & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 4)  & (v1eInit' = 1);
        [] (v1eInit = 0) -> (v1e' = 5)  & (v1eInit' = 1);
endmodule

module initv2e = initv1e [v1e = v2e, v1eInit = v2eInit] endmodule

module initw
        [] (wInit = 0) -> 0.00720808394646841 : (w' =   1) & (wInit' = 1) +
                          0.00729191001115681 : (w' =   2) & (wInit' = 1) +
                          0.0073755584209887  : (w' =   3) & (wInit' = 1) +
                          0.00745900085035024 : (w' =   4) & (wInit' = 1) +
                          0.00754220874803646 : (w' =   5) & (wInit' = 1) +
                          0.00762515335264172 : (w' =   6) & (wInit' = 1) +
                          0.007707805708289   : (w' =   7) & (wInit' = 1) +
                          0.0077901366806846  : (w' =   8) & (wInit' = 1) +
                          0.00787211697349145 : (w' =   9) & (wInit' = 1) +
                          0.0079537171450014  : (w' =  10) & (wInit' = 1) +
                          0.00803490762510252 : (w' =  11) & (wInit' = 1) +
                          0.00811565873252124 : (w' =  12) & (wInit' = 1) +
                          0.00819594069233258 : (w' =  13) & (wInit' = 1) +
                          0.00827572365371708 : (w' =  14) & (wInit' = 1) +
                          0.00835497770795787 : (w' =  15) & (wInit' = 1) +
                          0.00843367290665765 : (w' =  16) & (wInit' = 1) +
0.00851177928016469 : (w' =  17) & (wInit' = 1) +
0.00858926685618736 : (w' =  18) & (wInit' = 1) +
0.00866610567858883 : (w' =  19) & (wInit' = 1) +
0.00874226582633714 : (w' =  20) & (wInit' = 1) +
0.00881771743260257 : (w' =  21) & (wInit' = 1) +
0.00889243070397935 : (w' =  22) & (wInit' = 1) +
0.00896637593981819 : (w' =  23) & (wInit' = 1) +
0.00903952355165172 : (w' =  24) & (wInit' = 1) +
0.00911184408269317 : (w' =  25) & (wInit' = 1) +
0.00918330822739516 : (w' =  26) & (wInit' = 1) +
0.009253886851045   : (w' =  27) & (wInit' = 1) +
0.0093235510093839  : (w' =  28) & (wInit' = 1) +
0.0093922719682264  : (w' =  28) & (wInit' = 1) +
0.00946002122306727 : (w' =  30) & (wInit' = 1) +
0.00952677051865214 : (w' =  31) & (wInit' = 1) +
0.00959249186849526 : (w' =  32) & (wInit' = 1) +
0.00965715757432889 : (w' =  33) & (wInit' = 1) +
0.00972074024545805 : (w' =  34) & (wInit' = 1) +
0.00978321281801037 : (w' =  35) & (wInit' = 1) +
0.0098445485740542  : (w' =  36) & (wInit' = 1) +
0.00990472116057367 : (w' =  37) & (wInit' = 1) +
0.00996370460827489 : (w' =  38) & (wInit' = 1) +
0.0100214733502111  : (w' =  39) & (wInit' = 1) +
0.0100780022402017  : (w' =  40) & (wInit' = 1) +
0.010133266571033   : (w' =  41) & (wInit' = 1) +
0.0101872420924159  : (w' =  42) & (wInit' = 1) +
0.0102399050286902  : (w' =  43) & (wInit' = 1) +
0.0102912320962463  : (w' =  44) & (wInit' = 1) +
0.0103412005206602  : (w' =  45) & (wInit' = 1) +
0.0103897880535107  : (w' =  46) & (wInit' = 1) +
0.0104369729888724  : (w' =  47) & (wInit' = 1) +
0.0104827341794593  : (w' =  48) & (wInit' = 1) +
0.0105270510524079  : (w' =  49) & (wInit' = 1) +
0.01056990362468    : (w' =  50) & (wInit' = 1) +
0.0106112725180696  : (w' =  51) & (wInit' = 1) +
0.0106511389738006  : (w' =  52) & (wInit' = 1) +
0.0106894848666954  : (w' =  53) & (wInit' = 1) +
0.0107262927189037  : (w' =  54) & (wInit' = 1) +
0.0107615457131754  : (w' =  55) & (wInit' = 1) +
0.0107952277056612  : (w' =  56) & (wInit' = 1) +
0.0108273232382323  : (w' =  56) & (wInit' = 1) +
0.0108578175503009  : (w' =  57) & (wInit' = 1) +
0.0108866965901316  : (w' =  59) & (wInit' = 1) +
0.0109139470256325  : (w' =  60) & (wInit' = 1) +
0.0109395562546102  : (w' =  61) & (wInit' = 1) +
0.0109635124144827  : (w' =  62) & (wInit' = 1) +
0.0109858043914368  : (w' =  63) & (wInit' = 1) +
0.0110064218290202  : (w' =  64) & (wInit' = 1) +
0.011025355136159   : (w' =  65) & (wInit' = 1) +
0.0110425954945964  : (w' =  66) & (wInit' = 1) +
0.0110581348657331  : (w' =  67) & (wInit' = 1) +
0.0110719659968771  : (w' =  68) & (wInit' = 1) +
0.0110840824268839  : (w' =  69) & (wInit' = 1) +
0.0110944784911861  : (w' =  70) & (wInit' = 1) +
0.0111031493262088  : (w' =  71) & (wInit' = 1) +
0.01111009087316    : (w' =  72) & (wInit' = 1) +
0.0111152998811963  : (w' =  73) & (wInit' = 1) +
                              0.0111187739099629  : (w' =  74) & (wInit' = 1) +
                              0.0111205113314961  : (w' =  75) & (wInit' = 1) +
                              0.0111205113314959  : (w' =  76) & (wInit' = 1) +
                              0.0111187739099629  : (w' =  77) & (wInit' = 1) +
                              0.0111152998811963  : (w' =  78) & (wInit' = 1) +
                              0.01111009087316    : (w' =  79) & (wInit' = 1) +
                              0.0111031493262088  : (w' =  80) & (wInit' = 1) +
                              0.0110944784911864  : (w' =  81) & (wInit' = 1) +
                              0.0110840824268835  : (w' =  82) & (wInit' = 1) +
                              0.0110719659968771  : (w' =  83) & (wInit' = 1) +
                              0.0110581348657329  : (w' =  84) & (wInit' = 1) +
                              0.0110425954945966  : (w' =  85) & (wInit' = 1) +
                              0.0110253551361593  : (w' =  86) & (wInit' = 1) +
                              0.0110064218290199  : (w' =  87) & (wInit' = 1) +
                              0.0109858043914368  : (w' =  88) & (wInit' = 1) +
                              0.0109635124144827  : (w' =  89) & (wInit' = 1) +
                              0.0109395562546102  : (w' =  90) & (wInit' = 1) +
                              0.0109139470256325  : (w' =  91) & (wInit' = 1) +
                              0.0108866965901318  : (w' =  92) & (wInit' = 1) +
                              0.0108578175503007  : (w' =  93) & (wInit' = 1) +
                              0.0108273232382323  : (w' =  94) & (wInit' = 1) +
                              0.0107952277056614  : (w' =  95) & (wInit' = 1) +
                              0.0107615457131754  : (w' =  96) & (wInit' = 1) +
                              0.0107262927189036  : (w' =  97) & (wInit' = 1) +
                              0.0106894848666955  : (w' =  98) & (wInit' = 1) +
                              0.0106511389738004  : (w' =  99) & (wInit' = 1) +
                              0.0106112725180697  : (w' = 100) & (wInit' = 1);
endmodule
```