Analyzing Old Games with Modern Techniques: Probabilistic Model Checking Using SCOOP and PRISM

Vincent Bloemen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
v.bloemen@student.utwente.nl

ABSTRACT

Model checkers can be used to solve mathematical problems, without the need of having prior knowledge on the technical solution to those problems. This paper focuses on the practical usage of the tool SCOOP, which uses several optimization techniques for state space reduction in probabilistic automata. SCOOP can be used in collaboration with probabilistic model checkers such as PRISM to solve a given probabilistic problem, so that a manual approach is unnecessary. Three case studies are examined to show how to use SCOOP and, more importantly, to give an understanding of when to choose a model checking approach instead of trying to calculate a solution manually. The case studies are specifically focused on probabilistic games because of the ease of understanding and the freedom of applying variations. The N-Player ruin, Yahtzee and the Game of the Goose are the games that are analyzed. Scaling and other alterations are applied to find out to what extent these affect the state space and time used as well as the mathematical interpretation. The usage of SCOOP's optimization techniques is investigated to find out if and how these reduce the state space and time used to solve the problems more efficiently. The results will show that in some cases, a model checking approach provides results that are not realistically obtainable with alternative techniques.

Keywords

Probabilistic model checking, Probabilistic Automata, SCOOP, PRISM, Probabilistic games

1. INTRODUCTION

A probabilistic model checker is used to analyze a model by testing its probabilistic behavior. When computing answers for mathematical problems, a model checking approach is probably not the first thing that comes to mind. Usually, solving a probabilistic problem consists of designing a formula that a mathematical tool uses to solve the problem. However, for some problems it is hard to find a correct formula. When using a model checking approach, constructing a correct specification of the problem and a formal definition of the question is sufficient to compute answers. A probabilistic model checker is used for this computation.

17th Twente Student Conference on IT June 25th, 2012, Enschede, The Netherlands.

The tool SCOOP [9] is used to optimize specifications (formal representations) for probabilistic models and to either generate the state spaces or export the optimized model to PRISM [3]. Several techniques (see Section 2) are used that can drastically decrease the time used for model checking by reducing the, to be generated, state space (the set of all possible states for the system). The optimized specification can then be analyzed with model checking tools such as PRISM. Although other model checkers exist, these are not used in this research (see Section 3). It is important to note that these state space reductions happen before the actual state space is generated. Another benefit of SCOOP is the neatness of the input language that it uses for constructing specifications: prCRL [4].

Various probabilistic problems have already been analyzed with model checkers, for instance analyzing the network flow in a randomized network protocol. However, there are still numerous examples in existence for which it is unclear if and how a model checking approach can provide solutions.

A set of probabilistic games will be investigated with the SCOOP tool to examine how each problem is modeled and solved. Games are chosen to be examined because they are easy to understand and variations can also be applied without having to change a lot to the original problem. The mathematical difficulty, scale of the problem and the optimization possibilities for these problems will be discussed throughout the paper. Knowing that the 'mathematical difficulty' cannot be objectively discussed, explanations of a mathematical approach and related work should give an idea of this difficulty.

The following example illustrates on how to analyze games with the SCOOP tool.

Gambler's Ruin Problem.

Consider a gambler starting with an initial fortune of d dollars. In each round (assume that the fortune is larger than 0), the gambler can either win or lose one dollar with respective probabilities p and *1-p*. The question is: what is the probability that the gambler reaches a fortune of N dollars before going broke?

Assume that for one instantiation of this problem, the initial fortune is two dollars, the probability to win a dollar is 0.6 and the fortune to be reached is five dollars.

The problem can be analyzed using the SCOOP tool, by modeling the situation as as depicted in Figure 1.

The specification is briefly explained in Section 2. To calculate the probability of reaching fortune N, the prCRL

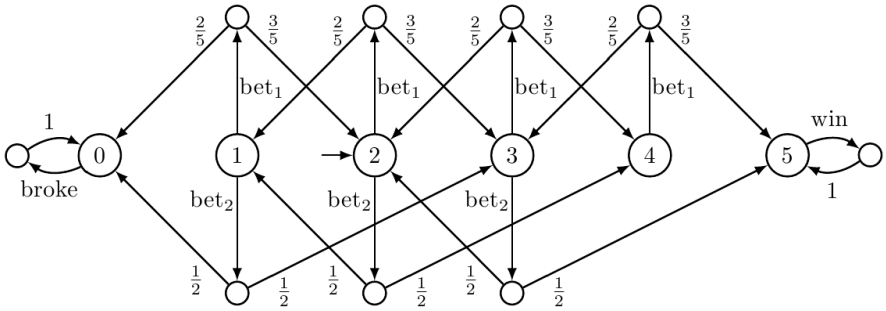


Figure 2. State space of the 'extended' Gambler's Ruin Problem

```
until [ true U win ]
constant N = 5
constant d = 2
constant p = 0.6
type Fortune = {0..N}

X(f:Fortune) =
    f=N => win.X[f]
 ++ f=0 => broke.X[f]
 ++ f>0 & f<N => bet.psum( p   -> X[f+1]
                         ++ 1-p -> X[f-1])
init X[d]
```

Figure 1. A prCRL specification for the Gambler's Ruin Problem.

model can be converted to a PRISM model. The probability to reach five dollars from the initial state can easily be computed in PRISM by calculating the probability to reach a 'win' action. The answer, 0.6398 can be computed (note that the probability is calculated, not simulated).

For this example, there is also an easy mathematical solution that yields the same result, a general case formula is provided in Equation 1. However, this example is specifically used to demonstrate a model checking approach in SCOOP and PRISM.

$$P_d = \begin{cases} \frac{1-\left(\frac{1-p}{p}\right)^d}{1-\left(\frac{1-p}{p}\right)^N} & , p \neq \frac{1}{2} \\ \frac{d}{N} & , p = \frac{1}{2} \end{cases} \quad (1)$$

Noteworthy is that in the model checking approach, no actual calculations had to be done by the user to solve the problem. One can assume that when problems get harder to solve mathematically, the model checking approach is a good alternative. When increasing the scale of the problem, though, the mathematical solution does not increase in difficulty while the model checking approach requires a larger amount of states to examine.

A nondeterministic choice can be added for the gambler; he may now choose between two types of 'betting'. The gambler's choices influence the outcome of the game. Nondeterminism implies that the player chooses an action with probability one. PRISM can deal with nondeterminism by differentiating between a *Pmin* and *Pmax*. These probabilities indicate the minimum and maximum probability of reaching the goal respectively.

Let bet_1 consist of the original case: one dollar is gained with probability 0.6 and one dollar is lost with probability 0.4. The gambler can now also choose to use the bet_2 option: two dollars are gained with probability 0.5 and one dollar is lost with probability 0.5. This bet option will only be applicable when the fortune is between one and $N-1$ dollars.

To incorporate this into the model, the following code is then added to the prCRL specification (above `init X[i]`):

```
 ++ f>0 & f<N-1 => bet2.psum(0.5->X[f+2]
                           ++ 0.5->X[f-1])
```

The resulting state space is depicted in Figure 2. The newly computed answer (with the use of *Pmax* in PRISM) is 0.6774, showing that the bet_2 option had a positive effect on the gambler.

This paper will focus on the practical usage of SCOOP by analyzing the N-Player ruin, Yahtzee and the Game of the Goose. These case studies are analyzed with SCOOP and PRISM, this approach is compared with alternative methods to show the potential of probabilistic model checking. Research questions are provided in the next section. This is followed by background information and related work. Then, each case study is analyzed and discussed. Finally, the research questions get answered in the conclusions.

1.1 Problem statement

The main question is as follows:

- To what extent are SCOOP and PRISM suitable for analyzing games compared to alternative methods, and what is the contribution of SCOOP's reduction techniques?

As shown in the Gambler's Ruin Problem, points of interest are the specification in the prCRL language and the 'mathematical difficulty' of the problem to be solved. Another interesting aspect is to find out the effects of SCOOP's optimization techniques on the analyzed games.

A reason for this investigation is to find out how prominent a model checking approach for solving probabilistic problems actually is, while also showing the potency of SCOOP in collaboration with PRISM. It is not yet completely clear when probabilistic model checking can be used effectively (and easily) to solve problems. Investigation with a number of case studies help to make this clear. There are only a few case studies for SCOOP in existence, making it a necessity to analyze new ones for showing SCOOP's potential.

1.2 Research questions

Based on the problem statement, the following research questions will be addressed:

- In which cases is a probabilistic model checking approach using SCOOP and PRISM easier than a mathematical computation?

To answer this, not only a specification and description of the analysis process will be given; insights in using a mathematical approach will be provided as well.

- How does the scale of the problem and complexity relate to the duration of the computation of answers?

In the Gambler's Ruin Problem, the scale can be increased by setting the number of dollars to be reached to a higher value. An example of a more complex situation is by adding the bet_2 option in the Gambler's Ruin Problem. Variances in these two aspects help to give an idea of where a mathematical approach encounters problems while also showing how the model checking approach handles this.

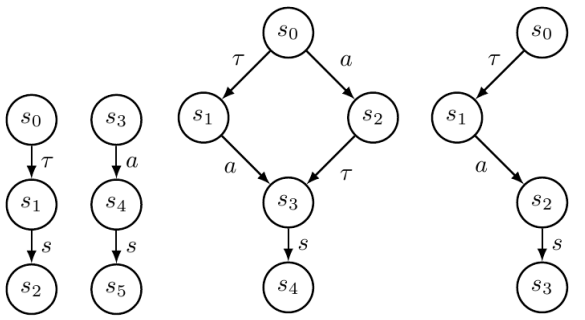
- How do SCOOP's optimization techniques influence the execution time needed for analyzing games?

Because SCOOP provides techniques that reduce the state space of a probabilistic model, it is useful to show how well these reductions work for the case studies. This is mostly done to validate the possibilities of SCOOP, attempting to show how usage of SCOOP can be beneficial in terms of execution time.

2. BACKGROUND INFORMATION

2.1 (Probabilistic) Automata

A non-probabilistic automaton consists of a finite set of *states* and *transitions* with one initial or 'starting' state. Transitions exist between two states, which do not have to be different, and require a *symbol* (also called label) to jump from one state to another. As shown in Figure 3(a), the process on the right can jump from s_3 to s_4 by executing an 'a' symbol. The special symbol τ defines an 'invisible' action (an action that has been hidden). Because τ is an invisible action, the process can act like it is in multiple states at once when a τ-transition is enabled. Two or more automata can be combined (*parallelized*). Equivalent actions are merged such as the s-transition in Figure 3(a), the parallelization of this is shown in Figure 3(b).



(a) Automata (b) Parallelization (c) Confluence reduction

Figure 3. Example automata

A probabilistic automaton (PA) is basically an automaton where probabilities are introduced to choose between multiple states. Figure 2 from the Gambler's Ruin Problem gives an example of this: from the (initial) state, 2, the symbol bet_1 is executed to jump with probability 0.6 to state 3 and with probability 0.4 to state 1.

2.2 SCOOP

SCOOP [9] is a tool that acquires specifications in a modeling language called prCRL. These specifications are then converted into an LPPE (linear probabilistic process equation), a restricted form of process equations. The process named X from the specification in Figure 1 is an example of a process equation. In the LPPE form, optimizations will be applied. These optimizations include basic reductions and dead variable reduction [11], and can reduce the state space. A smaller state space implies that fewer states need to be checked, which generally leads to a shorter execution time. Then, a state space can be generated. Confluence reduction [10] can be applied during this generation. The prCRL language is discussed next, followed by explanations of the reduction techniques.

2.2.1 The prCRL language

The prCRL language is a probabilistic version of the μCRL language [2], which is a process algebraic language that was especially developed to take into account data in the study of communicating processes. The prCRL language is intuitive when defining constants or data types, as shown in the Gambler's Ruin Problem. Some syntactical sugar can be used to enhance the ease of use as well.

Using the Gambler's Ruin Problem as an example, the starting state is given by `init X[d]`, where `X` is the process with the current fortune as a parameter. A nondeterministic choice is denoted with `++`, however, conditions can be used to prevent actions from taking place (e.g. only perform a `win.X[f]` if the fortune is equal to N). An action consists of a label and the outgoing probability distribution for the transition. A `psum` creates a 'probabilistic transition'; the next state will be `X[f+1]` with probability p and `X[f-1]` with probability $1 - p$. The `until` statement indicates that the model checker should look for the `win`-action and calculate for example the probability to reach this label. A detailed explanation of the prCRL language can be found at `http://wwwhome.ewi.utwente.nl/~timmer/scoop/syntax.html`, a formal description of prCRL is given in [4].

2.2.2 Basic reductions

The basic reductions [4] consists of *constant elimination* (parameters that always have the same value are replaced by this value), *summation elimination* (nondeterministic choices for which only one of the alternatives is applicable are simplified) and *expression simplification* (by use of logics and evaluation of functions, conditions and parameters can be rewritten). While not affecting the state space, these techniques can provide significant time reductions when building a model.

2.2.3 Dead variable reduction

Dead variable reduction [4, 11] reduces the state space while preserving strong probabilistic bisimulation (this is when two processes cannot be distinguished by an observer, even though τ is treated as a 'normal' action). This technique tries to find states in which the value of a global variable is irrelevant, for example when a variable will always be overwritten in the future before actually being used. In these states, the variable will be reset to its initial value to reduce the amount of states. When using large data types, these reductions work best due to the significant effect of resets.

2.2.4 Confluence reduction

Confluence reduction [10] detects internal τ-transitions that can be used to construct a smaller state space while

preserving a branching probabilistic bisimulation. This is basically strong probabilistic bisimulation with more freedom concerning the occurrence and order in which τ-transitions happen. When confluence reduction is applied to Figure 3(b), the state space of Figure 3(c) can be obtained because it does not matter when the τ-transition takes place. The s_0-state can be removed as well because the occurrence of τ is not monitored. When dealing with many τ-transitions, obtained by *parallel composition* (executing multiple processes at one time) and *hiding* (renaming a labeled transition to a τ-transition), confluence reductions can decrease the state space significantly.

2.3 Model checking

A prCRL model can be converted into a PRISM model. For this conversion, no state space is generated so confluence reduction cannot be applied. The basic reductions and dead variable reduction can be applied for obtaining a PRISM model. Besides the probability of reaching a specific situation, by using a *reward* (a predefined value for every occurrence of a label), the expected time to reach this specific situation can be found as well in PRISM. As explained in the Gambler's Ruin Problem, PRISM also supports *Pmin* and *Pmax* for finding probabilities in a nondeterministic model, by choosing the actions that result in the minimum and maximum probability respectively.

3. RELATED WORK

A few case studies can be found that make use of SCOOP. These include variations on a 'leader election protocol' [9, 10, 4], 'A perfect strategy for roulette' and 'Knuth's die'[1]. These case studies will not be used because of redundancy.

PRISM [3] could also be used for specifying a model, so omitting SCOOP completely. However, PRISM cannot make use of SCOOP's optimizations. The input language for SCOOP is also more generic.

PROBMELA [1] can also be used for probabilistic model checking in a similar way as SCOOP is used. PROBMELA, however, only supports basic data types and it does not provide comparable optimizations.

PRISM also provides a wide range of case studies concerning probabilistic model checking[2], which range from communication protocols to biological processes. Most of these case studies also provide a description of the specification that is used for solving the problem. Scaling is applied in some of the case studies with comparisons of the resulting state spaces. Because the case studies are either unnecessarily complicated (for the purpose of this research) or discussing a completely different field of study, an appropriate set of problems is created. Another reason for not considering these case studies is that they have already been analyzed; the existing solution can be modeled in SCOOP without much prior knowledge of the problem. The creation process for a specification is important for this research purpose, so insights can be provided in the difficulty of the modeling process.

Non-probabilistic problems can also be solved with model checkers, as discussed in Shilov *et al.* [7]. This paper explains the use of model checking tools for game-based constraints, by presenting a model checking based solution for a complicated puzzle.

[1]SCOOP case studies: `http://wwwhome.ewi.utwente.nl/~timmer/scoop/casestudies.html`

[2]PRISM case studies: `http://www.prismmodelchecker.org/casestudies/index.php`

For the specific case studies, related work is discussed in the concerning sections.

4. CASE STUDIES

Three case studies are discussed in this section, namely the N-Player ruin, Yahtzee and Game of the Goose. Each case study is structured to first state the problem to investigate; the 'game' is defined, and requested probabilities and expectations are stated. This is followed by a mathematical approach to the problem (by referring to existing work and/or explaining the difficulty). Then, a prCRL model is constructed for the problem, that will be thoroughly explained. The specified model will then be analyzed with PRISM, and answers to the problem are provided if possible. Scaling is applied to the model to show the increase in state space and time used, while comparing a model in which reductions are applied to one without reductions. Conclusions can be provided from the obtained results of each case study. The constructed models can be found at `http://wwwhome.cs.utwente.nl/~timmer/scoop/vincent/`.

Note that the used machine for the case studies has a 1.83 GHz Intel(R) Core(TM)2 Duo CPU, 3.00 GB RAM and it runs on a 32-bit Windows 7 Operating System.

4.1 N-Player ruin

Problem definition.

Using the Gambler's Ruin Problem that is discussed in Section 1 as a basis, a multiple-player version is considered. Swan [8] and Rocha *et al.* [6] provide examples that extend the Gambler's Ruin Problem to make it suitable for multiple players.

One example states that, considering N players that all start with an initial fortune of d dollars, one player is selected randomly with probability p_i ($\sum_{k=1}^{N} p_k = 1$ and not every p_i is the same) to win one dollar from each other player. The winning player gains $N-1$ dollars while each other player is set back one dollar. The game is over when one player is broke. This example is also known as the *asymmetric* N*-player Gambler's Ruin Problem with equal initial fortunes.*

This version is chosen to be analyzed. Two aspects of the problem are considered: the probability that a given player goes broke (referred to as the individual ruin probability), and the expected time until someone goes broke (the expected game length).

Figure 4 provides a visual representation for the case that $N = 3$ and $d = 3$, the probabilities for each transition are omitted. Each state represents the concatenated fortunes of player 1,2 and 3 respectively. From the starting state, 333, if player 1 wins, he gains two dollars while the others lose one dollar, so the next state would then be 522. Every state that contains a 0 has no outgoing transitions, because at least one player is broke and thus the game ends.

A mathematical approach.

In related studies, explicit formulas have been provided to solve specific cases of the problem. The individual ruin probability and expected game length is known when $N = 2$, for all initial fortunes d. A similar formula is provided in the Gambler's Ruin Problem (Section 1).

Rocha *et al.* [5] have shown that the expected game length, as well as the individual ruin probability, can be calculated directly whenever $0 \leq d \leq N + 1$, for $N \geq 3$.

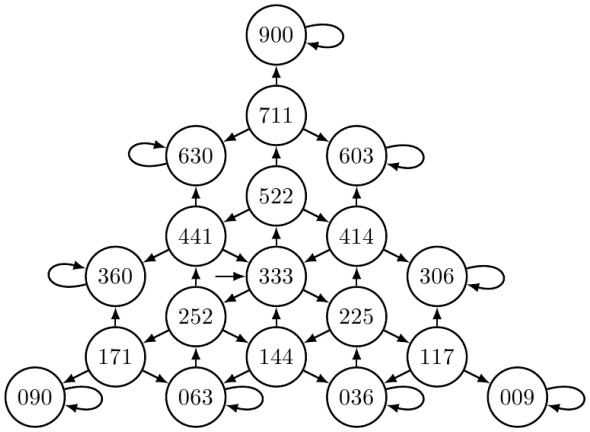


Figure 4. Simplified state space of the *N*-Player Ruin with $N = 3$ and initial fortune $d = 3$.

Rocha *et al.* [6] discuss the cases in which $d = N + 2$ and $d = N + 3$, for all $N \geq 3$, and provide explicit formulas for both the expected game length and the individual ruin probability. To give an idea of the mathematical complexity, the expected game length for $d = N + 2$ is calculated as follows:

$$E(T) = 2 + N\left(1 + \alpha\beta(N+2)(N+1)/(4!\Delta)\right),$$

for $\alpha = n!p_1p_2\ldots p_N$, $\beta = 3 - \alpha + p_1^2 + \ldots + p_N^2$ and

$$\Delta \equiv 1 - \alpha(N^2 + 3N + 6)/8 + \alpha^2(N+2)(N+1)/4!.$$

Swan [8] and Rocha *et al.* [6] both conclude that for values of $d = N + r, r > 3$ the length of the formula grows exponentially.

Providing prCRL specifications.

To examine this problem using model checking, a prCRL specification is constructed. There are multiple specifications that correctly represent the model. The probability to win a round needs to be provided for each player. These values are chosen as follows: $p_1 = 0.5$, $p_2 = 0.4$ and $p_i = \frac{1}{10(N-2)}$ for $3 \leq i \leq N$. Note that $\sum_{k=1}^{N} p_k = 1$.

One possibility is to model each player's fortune as a parameter for a 'main' process. This process will then choose player i with probability p_i as a winner and adds $N-1$ dollars to his fortune. Simultaneously, one dollar is removed from every other player's fortune. When one of the players is broke, the process should 'end'. The specification in Figure 5 is constructed based on this approach.

The *MF* constant represents the maximum fortune that a player can have when the game has not finished yet. The constraints that check if each player's fortune does not exceed *MF* is needed for PRISM (even though this cannot happen in practice). PRISM will otherwise think that players can obtain a fortune of larger than $N \cdot d$.

Considering scaling possibilities, alterations to d do not require modifications to the specification. But when N is changed, the parameters and constraints of the process need to be changed. For large values of N this approach is somewhat cluttered. This approach is referred to as *the static approach*, due to the amount of work for scaling the model.

Another approach is by using parallel composition, showed in Figure 6. This approach will be called *the dynamic approach*.

```
until [ true U broke ]

constant N = 3
constant d = 3
constant p1 = 0.5
constant p2 = 0.4
constant pi = 1/(10*(N-2))
constant MF = N*d-(N-1)

type Fortune = {0..N*d}
X(f1:Fortune, f2:Fortune, f3:Fortune) =
   f1=0 | f2=0 | f3=0 =>broke.X[f1,f2,f3]
++ f1>0 & f2>0 & f3>0
   & f1<=MF & f2<=MF & f3<=MF =>
     bet.psum( p1 -> X[f1+N-1,f2-1,f3-1]
             ++ p2 -> X[f1-1,f2+N-1,f3-1]
             ++ pi -> X[f1-1,f2-1,f3+N-1])

init X[d,d,d]
```

Figure 5. The static approach for the *N*-Player Ruin with $N = 3$ and initial fortune $d = 3$.

```
until [ true U broke ]

constant N = 3
constant d = 3
constant MF = N*d-(N-1)

type Fortune = {0..N*d}
X(i:{1..N}, f:Fortune) =
   f=0 => broke.X[i,f]
++ f>0 => bet.(f<=MF =>step(i).X[i,f+N-1]
      ++ sum(j:{1..N}, j!=i & f>0
             => step(j).X[i,f-1]))

S = bet.psum(k:{1..N},
       ifthenelse(eq(k,1), 0.5,
       ifthenelse(eq(k,2), 0.4,
       1/(10*(N-2)))) : step(k).S[])

init S || X[1,d] || X[2,d] || X[3,d]
nocomm broke
```

Figure 6. The dynamic approach for the *N*-Player Ruin with $N = 3$ and initial fortune $d = 3$.

Every player is modeled as a process X with an index and fortune as parameters. A system process, S, repeatedly chooses a winner by performing a *step(k)* action. The corresponding player will then increase its fortune by $N - 1$ dollars and others' fortunes decrease by one dollar. This is because the action *step(k)* must be executed simultaneously in each process (parallel composition). When a player goes broke, he is only able to execute the *broke* action. The command *nocomm broke* makes sure that this action does not have to be synchronized (the system process does not have to execute the broke action).

In this approach, scaling is much easier. When increasing N, new processes should be included in the *init*.

Analyzing the model.

The specification is analyzed with PRISM. In the resulting PRISM model, a reward of 'one' is applied for each time that a *bet* action occurs for finding the expected game length. The previously stated *until* formula is now applied in the model so that the system is *satisfied* when the first broke action occurs. The expected game length can then be calculated. The individual ruin probability for player one can be found by changing the *until* formula to stop

Table 1. Analyzing the N-Player Ruin with initial fortune d, - means out of memory.

		The static approach			The dynamic approach			Expected	Ruin
			Time used (s)			Time used (s)		game	probability
N	d	Nr of states	Original	Reduced	Nr of states	Original	Reduced	length	(player 1)
3	3	28	0.048	0.048	53	0.153	0.106	3.409	0.142
3	10	196	0.090	0.136	585	0.430	0.210	14.115	$3.77 \cdot 10^{-3}$
3	50	3,976	1.866	1.610	14,905	13.387	6.892	71.429	$1.15 \cdot 10^{-11}$
3	100	15,451	11.631	10.260	59,805	67.918	35.020	142.857	$3.13 \cdot 10^{-22}$
3	500	377,251	575.893	574.196	1,499,005	4476.569	2937.857	714.286	$1.81 \cdot 10^{-106}$
5	5	376	0.178	0.141	937	-	0.359	5.004	0.0313
5	10	3,626	0.523	0.435	12,417	-	1.577	10.092	$1.00 \cdot 10^{-3}$
5	50	1,464,376	52.017	51.764	7,661,557	-	423.751	53.640	$3.46 \cdot 10^{-15}$
5	100	21,449,977	2583.542	2570.123	123,646,857	-	-	110.541	$2.25 \cdot 10^{-29}$
10	5	5,005	0.483	0.614	14,300	-	3.569	5.000	0.0313
10	10	277,132	3.921	4.510	1,152,293	-	17.723	10.000	$9.77 \cdot 10^{-4}$
10	15	4,571,259	89.274	92.381	23,358,621	-	343.564	15.000	$3.05 \cdot 10^{-5}$
10	20	39,782,886	-	-	232,344,684	-	-	-	-

when a *broke(1)* action occurs, a *broke(i)* action should only be possible if player i is broke.

Table 1 provides insights in the scaling possibilities of both approaches. The amount of states and time used (in seconds) to find the expected value is used as an indicator (the individual ruin probability provides similar results). It can be seen that for small values for N and d, both approaches provide almost immediate results whereas large values take significantly longer to calculate. The (reduced) dynamic approach is roughly four times slower than the static approach. This can be explained by the same difference in the number of states (which is because the static approach is modeled more efficiently).

The measured time is the sum of the construction time for the PRISM model, the time to *build* the model in PRISM and the time for model checking (computing the answer). To give an idea of the distribution of these times, the case that $N = 5$ and $d = 50$ is examined for the reduced static approach: the construction time for the PRISM model accounted for 141 milliseconds, the build time for the PRISM model accounted for 5.393 seconds and the time for model checking accounted for 46.230 seconds. These distributions are similar for the other cases.

Both approaches are analyzed with unoptimized and optimized models. Basic reductions were omitted in the *original* models. Dead-variable reduction was applied on top of the basic reductions in the *reduced* models. It should be noted that SCOOP's reductions were not able to reduce the state spaces, but the PRISM models are constructed more efficiently, and this significantly reduced the used time in the dynamic approach. For $N \geq 5$, the constructed PRISM models for the dynamic approach are too large for PRISM to handle, so these cases cannot be analyzed.

The resulting probabilities and expectations for $d \leq N$, for all values of N, are verified by applying the equations stated in Rocha *et al.* [5].

Conclusions.

The N-Player Ruin can be correctly modeled in SCOOP using multiple approaches. For $3 \leq d, N \leq 10$, the expected game length and the individual ruin probability are solved in less than 10 seconds, in both discussed approaches. For larger values of either d or N, the computation time increases rapidly. Mathematical approaches scale better for a larger value of N as there are explicit formulas for finding the stated probabilities. However, when d increases and surpasses N, the mathematical approach starts getting computationally heavy. The conclusion can therefore be made that a probabilistic model checking approach is easier than a mathematical computation, especially in the case that N is small and d surpasses N (for example $N = 5$ and $d = 10$).

4.2 Yahtzee

Problem definition.

The game of Yahtzee[3] can be played alone. One round of the game starts with the player throwing five dice. The player can then choose to *lock* some or all dice and throw again with the remaining dice. This process is thereafter repeated once more. The resulting combination is then used to fill in a scoreboard. In total, there are 13 rounds and scoring categories that can be selected. The goal is to score as much points as possible.

For this case study, one round of Yahtzee is considered, and the probability of achieving *Yahtzee* (five dice with the same value) is going to be analyzed. Also, the expected number of turns needed before rolling Yahtzee will be analyzed. Probabilities for other scoring categories will be discussed, and the number of dice and turns will be altered to provide insights in scaling possibilities.

A mathematical approach.

Related work [12, 13] has successfully approximated the probabilities for each outcome. Moreover, the optimal strategy for playing the game has been approximated. Verhoef *et al.* [12] achieved this by simulating one million games that were used to approximate the probabilities.

The exact probability of achieving *Yahtzee* can be found by using a Markov Chain, `http://www.datagenetics.com/blog/january42012/index.html` discusses how this is manually calculated in detail. The resulting probability of getting Yahtzee is 4.603%, under the assumption that the dice with the most frequent value are locked. The same site discusses these probabilities when more turns are allowed. When for instance the player is allowed to throw 10 times, the probability of achieving *Yahtzee* is over 55.553%.

Providing a prCRL specification.

One round of Yahtzee can be modeled by tracking the values for the dice. A fairly straightforward specification

[3] *Yahtzee* is a registered trademark of Milton Bradley, which is owned by Hasbro.Inc.

of the model is given in Figure 7. This makes use of parallel composition by letting the *turn* action execute three times.

```
X(n:{1..5}, v:{0..6}) =
    turn.psum(k:{1..6}, 1/6 : X[n,k])
 ++ v > 0 => turn.X[]

S = turn.turn.turn.E[]
E = end.E[]

init S || X[1,0] || X[2,0] || X[3,0]
       || X[4,0] || X[5,0]
```

Figure 7. Straightforward approach on the Yahtzee model, using parallel composition.

In the model, n represents the index of each die and v represents the values that each die can take. Initially is v set to 0, so that the process knows that it is the first throw. Note that each die can only be thrown three times. Also, the player can choose to keep the value of a die, which adds a nondeterministic choice.

However, the resulting state space exceeds the memory limit and can therefore not be used for analyzing purposes (or at least not with the used computer specifications). Note that when the fifth die is removed from the game, the resulting state space is within the memory limit (3.889 states and 6.225.984 transitions).

Because it does not matter how the specific die values are configured (for example, $[4, 1, 2, 3, 2]$ is equivalent to $[1, 2, 2, 3, 4]$), it is possible to specify a more efficiently constructed model. Figure 8 represents such a model.

```
until [true U yahtzee]
constant MT = 5
constant N = 5
constant yahtzee = n1=N | n2=N | n3=N
                 | n4=N | n5=N | n6=N

X(n1:{0..N},n2:{0..N},n3:{0..N},n4:{0..N}
,n5:{0..N},n6:{0..N},d:{0..N},t:{0..MT})=
    yahtzee => yahtzee.X[]
 ++ d=0&mod(t,2)=0&t<MT => turn.X[t:=t+1]
 ++ mod(t,2)=1 & t<MT & d<N => (
         n1>0 => pick.X[n1:=n1-1,d:=d+1]
      ++ n2>0 => pick.X[n2:=n2-1,d:=d+1]
      ++ n3>0 => pick.X[n3:=n3-1,d:=d+1]
      ++ n4>0 => pick.X[n4:=n4-1,d:=d+1]
      ++ n5>0 => pick.X[n5:=n5-1,d:=d+1]
      ++ n6>0 => pick.X[n6:=n6-1,d:=d+1])
 ++ mod(t,2)=1 & t<MT => choose.X[t:=t+1]
 ++ d>0 & mod(t,2)=0 & t<MT &not(yahtzee)
    => roll.psum(
         1/6 -> X[n1:=n1+1,d:=d-1]
      ++ 1/6 -> X[n2:=n2+1,d:=d-1]
      ++ 1/6 -> X[n3:=n3+1,d:=d-1]
      ++ 1/6 -> X[n4:=n4+1,d:=d-1]
      ++ 1/6 -> X[n5:=n5+1,d:=d-1]
      ++ 1/6 -> X[n6:=n6+1,d:=d-1])

init X[0,0,0,0,0,0,N,0]
```

Figure 8. Second approach on the Yahtzee model

This model has some similar lines of code. In SCOOP it is possible to make use of arrays and remove this redundancy, but PRISM does not support arrays.

Instead of keeping track of each die, the number of occurrences for each value is 'stored' in the parameters of the process. The parameters $n_1, \ldots, n_6$ hold the number of dice that have the respective value $1, \ldots, 6$. The parameter N represents the total number of dice that are used in the game. The parameter d is used to keep track of the number of dice that still need to be thrown, which is initially set to 5. The parameter t shows the current 'turn', ranging from 0 to a the maximum amount of turns: MT.

A round of Yahtzee has the following set of sequential phases: $roll \rightarrow choose \rightarrow roll \rightarrow choose \rightarrow roll \rightarrow end$, these six stages are represented by t. Every even turn represents a *roll* and an odd turn represents a *choose* or *end*, MT should therefore be chosen as an odd number. Note that $t := t + 1$ indicates that only the parameter t is changed and the remaining parameters stay the same.

First is the *roll* phase: all d (initially N) dice will be rolled. The values for these dice are stored in $n_1, \ldots, n_6$. Then, the player selects dice to roll again with the *pick* action. For each chosen die is d (the number of dice to roll) increased by one and n_i is decreased by one for a die with value i. This process is repeated twice more.

In this approach the resulting state space is within the memory boundaries ($2,598$ states and $8,298$ transitions).

Analyzing the model.

The probability of achieving *Yahtzee* is found by checking when the until formula is satisfied. Noteworthy is that *Pmax* should be used in PRISM, as the player should make the most optimal choices (locking the dice with the most frequent value). The resulting probability is 4.603%, equal to the result of the mathematical approach.

When *Pmin* were to be chosen in the standard case (five dice and three turns), the 'least' optimal choices are made. In other words, the probability of reaching *Yahtzee* while trying to prevent this from happening. If the initial throw is not *Yahtzee*, the player will choose to not throw again and *Yahtzee* will never be achieved. If the initial throw is *Yahtzee*, the player will throw all dice again, and possibly a third time if *Yahtzee* is achieved once more. The probability of still reaching *Yahtzee* after three throws equals $((\frac{1}{6})^4)^3$. This same probability can be found with PRISM.

Probabilities for other categories can be calculated as well. The probabilities for a *Four of a Kind* or *Three of a Kind* are found using a similar property. A *Straight* (five consecutive values) is found by providing both cases in which a *Straight* can occur. A *Full House* can be found by checking if a *Pair* and a *Three of a Kind* occur simultaneously and are non-overlapping.

The expected number of turns needed for reaching *Yahtzee* can also be calculated. This does require some remodeling as the process should only end when *Yahtzee* is achieved. This model will not be discussed in detail. The expected amount of turns (note that in this case, one turn consists of both the *roll* and *choose* phases) needed for reaching Yahtzee is 11.0901. The player should make the most optimal choices in this case (similar to *Pmax*). With the worst possible choices, the expected amount of turns needed is infinite (as the player can lock two differently valued dice).

Table 2 shows how scaling affects the time needed (in seconds) for calculating the probability. The time used is measured as in Section 4.1. The probability to throw *Yahtzee* (all N dice have the same value) approaches one as the number of turns increases, which is obvious. The number of turns can increase to 1000 in the case that $N = 5$, the used time and state space increases linearly as MT increases. However, increments to N exponentially increase the state space and time used. In the case that $N = 40$

Table 2. Analyzing the scaling possibilities of Yahtzee, using N dice, - means out of memory.

N	turns	Nr of states	Time (original)	Time (reduced)	P{Yahtzee}
5	3	2,598	0.209	0.121	0.04603
5	10	9,150	0.296	0.160	0.55553
5	100	93,390	10.871	11.076	0.99999
5	1000	935,790	168.596	165.323	0.99999
10	3	43,079	1.006	0.988	0.00077
10	10	155,275	6.794	6.829	0.25053
10	100	1,597,795	249.351	253.722	0.99999
35	3	23,139,984	1050.778	1077.542	$4.21 \cdot 10^{-13}$
40	3	48,055,890	-	-	-

and $MT = 5$, it was possible to construct a PRISM model, but an out of memory error occurs when model checking for the answer.

For all values of N and MT, the original model (the model that did not undergo reductions) performed equally fast as the reduced model (no significant difference can be found). The number of states in the original model is equal to the number of states in the reduced model.

Conclusions.

Yahtzee cannot be modeled directly due to the large state space it generates. But when the game is modeled somewhat more efficiently, it is possible to calculate the stated probabilities. This approach is scalable to a certain degree, while still able to provide answers in time. SCOOP's reductions do not seem to benefit this time. Manually calculating probabilities for the standard case is possible, but is a lot of work when the number of turns or dice increases. Simulating the game is another possibility, however, to achieve a high level of precision, a large number of simulations is required. So, a model checking approach is a good alternative when a high precision is requested.

4.3 Game of the Goose

Problem definition.

The Game of the Goose (in Dutch: "Ganzenbord") is a board game that consists of 63 sequentially numbered squares (excluding the start square). The players alternately roll two dice that depict the number of squares that can be moved. The goal of the game is to reach square 63 exactly. When a player overshoots this value, remaining squares should be counted backwards from square 63 (the player resumes forward movement in the next turn). Furthermore, the following rules are defined[4]:

- Squares 5, 9, 14, 18, 23, 27, 32, 36, 41, 45, 50, 54 and 59 - *goose*: when a player lands on one of these square, he should move the same amount of the roll again (it is possible to land on multiple geese in one turn).
- Square 6 - *bridge*: the player moves to square 12.
- Square 19 - *inn*: the player needs to wait one turn.
- Squares 31 and 52 - *well*: the player cannot move until someone else reaches this square, the other player is then 'trapped' in the well and cannot move until he is 'saved'.
- Square 42 - *maze*: the player moves to square 37.
- Square 58 - *death*: the player moves to the starting square.
- Whenever a player reaches a square which contains another player's pawn, he should move back to his old position and end his turn.
- In the first round: when a player rolls a 4 and a 5, he moves to square 53. When a player rolls a 3 and a 6, he moves to square 26 (this is to prevent a player from starting with a 9, which would let him win the game in one turn). This will be known as the first round rule.

For this case study, the board game is modeled for two players. The probability that a given player wins, or any player at all (both players could become trapped in different wells, this results in a tie), will be analyzed. The expected game length in rounds will also be determined.

A mathematical approach.

For this game, there is no direct way of telling what the winning probability is. This is because for each turn, every die value needs to be observed, and the opponent's position has to be known as well. These possibilities, alongside the rules and restrictions of the game, result in a realistically impossible calculation.

The system therefore has to be modeled in an efficient manner. If a model checking approach is not chosen, another method providing answers is to simulate the game. This has been done (on a slightly simplified version of the game) using a Haskell program: `http://norsetto.site11.com/?p=25`. The probability that a game ends after a given amount of turns was approximated in this simulation for a four player version. They found that the probability for a game length of less than 32 steps is about 72%.

Providing a prCRL specification.

To construct a model for this game, a simplistic version is considered that will gradually become the complete game. All special squares and restrictions are not considered yet. A model for this situation is provided in Figure 9.

Here, $n1$ and $n2$ represent the current squares for both players and t indicates whose turn it is. The two dice are modeled as i and j for player one and as k and m for player two. After each roll, the turn is swapped and measures are taken when a player overshoots square 63. Furthermore, the game ends when one player reaches square 63.

However, when this specification is adapted to include all rules, a comprehensive specification is constructed. The complete specification is not provided, but individual aspects are discussed.

[4]The used rule set: `http://www.spelregels.eu/rules/ganzenbord/00314.pdf` (in Dutch, French and German).

Table 3. Analyzing a two-player version of the Game of the Goose.

Nr of dice	Special conditions	Nr of states	Original time	Reduced time	P{player one wins}	P{player two wins}	Expected game length
1	No first round rule	36,261	7.871	7.304	0.36758	0.34864	34.7532
1	No first round rule, no wells	37,017	4.882	4.464	0.51728	0.48272	42.4297
2	-	188,893	93.017	90.424	0.40776	0.40372	26.1828
2	No wells	195,795	78.714	79.727	0.50902	0.49098	29.7568
2	No first round rule, No wells	195,331	79.123	80.914	0.51302	0.48698	24.4997

```
until [ true U win ]

X(n1:{0..69}, n2:{0..69}, t:{1..2}) =
   n1=63 | n2=63 => win.X[]
++ t=1 & n1 < 63
=> roll_1.psum(i:{1..6}, 1/6
             || j:{1..6}, 1/6 :
     n1+i+j<=63 => X[n1+i+j,n2,2]
  ++ n1+i+j>63 => X[63-(n1+i+j-63),n2,2])
++ t=2 & n2 < 63
=> roll_2.psum(k:{1..6}, 1/6
             || m:{1..6}, 1/6 :
     n2+k+m<=63 => X[n1,n2+k+m,1]
  ++ n2+k+m>63 => X[n1,63-(n2+k+m-63),1])
init X[0,0,1]
```

Figure 9. A specification for a simple version of the Game of the Goose.

When a player is on a special square, for instance a *bridge*, specific actions need to be taken. For reading purpose, constants are created that define the situations on which a player is on a specific square. For player one, these constants are provided as displayed below.

```
constant p1_Goose = n1=5 | n1=9 | n1=14
  | n1=18 | n1=23 | n1=27 | n1=32 | n1=36
  | n1=41 | n1=45 | n1=50 | n1=54 | n1=59
constant p1_Bridge = n1=6
constant p1_Inn = n1=19
constant p1_Well = n1=31 | n1=52
constant p1_Maze = n1=42
constant p1_Death = n1=58
constant p1_Start = n1=0
constant p1_NotSpecial = not(p1_Goose
  | p1_Bridge | p1_Inn | p1_Well
  | p1_Maze | p1_Death)
```

The events that $n1 = n2$ and $n1 \neq n2$ are distinguished.

The $n1 = n2$ situation can only occur on the starting square and on a *well* (the first player that reached the *well* should move in the next turn). In other cases, the last player that entered should move back to his previous square. In order to reduce the total state space, instead of keeping track of each player's previous positions, only the number of squares that the player moved is tracked. New problems arise as this approach does not work when when the player traveled via a *bridge* or *maze*. To solve this problem, before a *bridge* or *maze* is applied, a check is made to see if the resulting square (respectively square 12 or 37) is free.

When $n1 \neq n2$ and it is the second player's turn, the cases in which player one is on a special square are treated. This is because when a player rolls the dice and moves to a square, the turn is transfered to the next player. This next player cannot 'begin' his turn when someone else is on a *goose*, *bridge*, *maze* or *death* square. If the other player is not on any of these special squares, he can be on the *well* square, *inn* square or the not special square. If he is on the *well* square, he cannot move so no action is taken. If he is on the *inn* square, a parameter w is used as a counter. The first time that player two is in the *inn*, the counter increases from zero to one. On the next round (in the case that $w = 1$) is player two free to move again and w is set back to zero. On a not special square, the player acts as if it were the described model from Figure 9.

Actions must be taken when a value of nine is rolled in the first round. To make sure that this is not necessary in other cases that a player is on the start square, a variable f is introduced. This variable is set to one initially, and changes to zero after player two has rolled his first dice. Only in the cases that $f = 1$ and nine is rolled is the rule applicable.

Analyzing the model.

The specification can be analyzed by adapting the *until* formula to represent the requested probability or expected value. The model is deterministic because no choices are made by the players.

Instead of scaling, some variations of the game are applied. A version with one die instead of two is considered, but the first round rule cannot be applied in this case. Because the two *wells* can create a deadlock situation (the players are in different *wells*). Noteworthy, the probability to reach the described deadlock situation is $1 - P\{\text{player one wins}\} - P\{\text{player two wins}\}$. By configuring the constants `p1_Well` and `p2_Well` to return false, the *wells* are omitted. The first round rule is omitted by setting the initial value for f to zero.

Table 3 displays the results. The state spaces for the cases that use two dice are roughly six times larger than the state spaces in which one die is used. A reason for this is that the values for the first and second die need to be known (in the first round). Because in the remaining rounds, only the sum of the two dice is needed, it can be questioned if a more efficient solution exists.

The probability that player one has an advantage over player two is explained by a number of reasons. The first player is one turn ahead, which gives him a natural advantage. Also, because a player needs to go back to his original position if he lands on the other player, player two is held up more. Another reason is that if player one were to end up in a *well*, the probability that player two reaches this *well* is larger than the other way around.

It is also noteworthy that the probability to reach a deadlock situation is 0.18852 in the standard (two dice and no special conditions) situation. This means that roughly one in every five games with two players ends in a tie. The probability that player one wins instead of player two is on average 2.0 percent point higher. In the standard sit-

uation this difference is less significant. A reason for this might be that in the first round, player two can roll the same as player one. Therefore he has to go back to the starting square, but when he now rolls nine the first round rule does not apply anymore. Player two would then go to square 63 by using the geese.

The time used is measured equally as done in Section 4.1. The reduction techniques do no reduce the state space nor improve the time used (the difference is not significant).

A three or four player version could be considered, but this does require a drastic change to the specification. A three player version with one die and no wells is constructed. It consists of 2,530,035 states and exceeds one hour when model checking. Results for the three player version are therefore not provided.

Conclusions.

The winning probabilities for this game are realistically impossible to solve manually due to the large amount of possible states. A simulation-based approach is possible, but only approximations can be given for the probabilities. When using a model checker, these probabilities can be calculated exactly for a two player version. However, the model needs to be specified efficiently to prevent the state space from growing too large. When the model is constructed, variations can be applied easily to restrict special squares. The reduction techniques of SCOOP do not benefit the results time-wise nor the amount of states.

5. CONCLUSIONS

The results from the three case studies are used to answer the research questions, stated in Section 1.2.

In which cases is a probabilistic model checking approach using SCOOP and PRISM easier than a mathematical computation?

When comparing a probabilistic model checking approach to a mathematical computation, there are definitely cases for which a model checker provides answers more easily. In the N-player ruin, the cases for which the initial amount of money exceeds the number of players is mathematically hard, while a model for this problem is fairly easy to specify. For Yahtzee and the Game of the Goose, modeling was more difficult. The specified models required more insight into the problem and the efficiency (state space it generates) had to be kept in mind as well. Still, once the model was specified, probabilities were computed very easily.

How does the scale of the problem and complexity relate to the duration of the computation of answers?

Scaling and variations to the problem were applied easily (only a constant value was modified in most cases). Obviously, when increasing the scale, the state space and computation time increases as well. If the original case already generated a large state space (for instance in the Game of the Goose), scaling to even larger models was not very feasible.

How do SCOOP's optimization techniques influence the execution time needed for analyzing games?

Even though SCOOP's reduction techniques have been shown before to benefit the state space and model checking time, these reductions did not benefit the time needed for analyzing Yahtzee and the Game of the Goose. Notably, because state space reductions were already applied when specifying the models, SCOOP's reductions could not further improve the efficiency. A reason for this are that confluence reduction could not be applied because confluence reduction can only be applied if SCOOP generates a state space, which is not the case. Also, there where no 'temporary' variables (so dead variable reduction could not help either). The N-player ruin does show the effect of the basic reductions. While not decreasing the state space, the used time and amount of memory needed for the model improved significantly.

To conclude from the results, probabilistic games can be solved effectively using SCOOP and PRISM. Especially when requesting accurate probabilities a model checking approach is more applicable then a simulation approach. For mathematically complex problems a model checking approach is also a good alternative. Another benefit is that once a model is specified, probabilities can be found for multiple situations (whereas a mathematical formula provides only one answer). It goes without saying that probabilistic model checking can be used effectively for solving a broad range of problems, not only game-specific.

This research has also been useful for the development of the SCOOP tool, as the functionality of parallel composition and the usage of constants in other constants (for instance $N = x$) has been added in the process of this research.

6. REFERENCES

[1] C. Baier, F. Ciesinski, and M. Größer. PROBMELA: a modeling language for communicating probabilistic processes. pages 57–66, 2004.

[2] J. Groote and A. Ponse. The syntax and semantics of μCRL. (CS-R9076), 1990.

[3] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. 3920:441–444, 2006. 10.1007/11691372_29.

[4] J. P. Katoen, J. C. van de Pol, M. I. A. Stoelinga, and M. Timmer. A linear process-algebraic format with data for probabilistic automata. *Theoretical Computer Science*, 413(1):36–57, January 2012.

[5] A. L. Rocha and F. Stern. The gambler's ruin problem with n players and asymmetric play. *Statistics & Probability Letters*, 44(1):87–95, 1999.

[6] A. L. Rocha and F. Stern. The asymmetric n-player gambler's ruin problem with equal initial fortunes. *Advances in Applied Mathematics*, 33(3):512–530, 2004.

[7] N. Shilov and K. Yi. Puzzles for learning model checking, model checking for programming puzzles, puzzles for testing model checkers. *Electronic Notes in Theoretical Computer Science*, 43(0):34–49, 2001. Formal Methods Elsewhere.

[8] Y. Swan. *On two unsolved problems in probability.* PhD thesis, Université Libre de Bruxelles, 2007.

[9] M. Timmer. SCOOP: A tool for symbolic optimisations of probabilistic processes. pages 149–150, September 2011.

[10] M. Timmer, M. I. A. Stoelinga, and J. C. van de Pol. Confluence reduction for probabilistic systems. 6605:311–325, March 2011.

[11] J. C. van de Pol and M. Timmer. State space reduction of linear processes using control flow reconstruction. 5799:54–68, October 2009.

[12] T. Verhoeff and E. Scheffers. Optimal solitaire yahtzee strategies. 2004.

[13] P. Woodward. Yahtzee:the solution. *Chance*, 16(1):17–20, 2002.