

Static Partial Order Reductions for Probabilistic Systems

Masterthesis

zur Erlangung des akademischen Grades
Master of Science (Ms.Sc.)

vorgelegt an der
**Technischen Universität Dresden,
Fakultät für Informatik**

eingereicht von Álvaro Fernández Díaz

Betreuende Hochschullehrerin:
Prof. Dr. Christel Baier

Dresden, im 2011



Abstract

The present Master's thesis seeks the development and analysis of static partial order reduction techniques for the models of probabilistic systems. The properties of those systems can be verified via model checking technique. Model checking suffers from the problem known as State Space Explosion, which can make the verification process intractable. Partial order reductions are aimed at alleviating that problem. As an outcome of the work carried out for the elaboration of current thesis, two new static partial order techniques, named Naïve SPOR and Reachability-Aware SPOR, were defined. A recommendation of the situations in which each of them should be used is provided. The latter achieves a better reduction than the former when the system to be verified is not probabilistic or when the property to be checked can be expressed in a linear temporal logic. The software tool known as LiQuor model checker was extended in order to be able to execute both reduction techniques. Those techniques were utilized for the reduction of the models of some classical concurrent systems. Several properties of the reduced and unreduced models were verified using symbolic and explicit model checking techniques. As a result of the analysis over the experiments, it is concluded that static partial order techniques should be more conveniently used in combination with symbolic model checking than with explicit model checking.

Acknowledgements

First of all, I want to thank Prof. Dr. Christel Baier for accepting me as her Master student since the very first moment I contacted her. I also thank Dr. Frank Ciesinski for his help during the development of this Master's thesis. He has always been friendly in the interaction and provided me with quick and very useful tips for the correct culmination of my work. I would like to thank Prof. Dr. Steffen Hölldobler for giving me the opportunity to finish my Master studies in Dresden. Last but not less important, I want to sincerely thank my PhD. supervisors in Madrid, Dr. Clara Benac Earle and Dr. Lars-Åke Fredlund. They informed me about the European Master's Program in Computational Logic (EMCL) for the first time, encouraged me to enroll in it and have supported me during the two years it has lasted. I owe them a lot of gratitude for their patience during my stay in Dresden and the trust on me that they always show.

Although their support has not been academical, I am sure that I could never have finished my Master's thesis without the reassurance of my friends in Spain, my colleagues from Babel Research Group in the Universidad Politécnica de Madrid and the new friends I have made during my stay in Dresden. I specially want to thank Beatriz and Dirk. Both of them listened patiently to my concerns when I got stuck with my work and encouraged me to finish it, always with a smile. They also allowed me to distract and freshen my mind during my leisure time through a whole bunch of great experiences. They have a well-deserved place deep within my heart. Finally, I wish to express my utmost gratitude to my parents Marcelino and Begoña. I owe them every step I take, as they keep constantly giving me the opportunity to accomplish all the dreams I could have never dared to dream of. My steps are also their steps.

0

Table of Contents

CHAPTER 1 – Introduction	1
<hr/>	
CHAPTER 2 – Basic Concepts	7
2.1. Notations	7
2.2. State Space Explosion Problem	11
2.3. Partial Order Reductions	14
2.3.1. Some POR Alternatives	18
<hr/>	
CHAPTER 3 – Towards Static POR for Probabilistic Systems	21
3.1. Sticky Set POR Technique	21
3.2. Sticky Set Identification: the Naïve Approach	22
3.2.1. Theoretically Expected Improvement	23
3.3. Experiments	25
<hr/>	
CHAPTER 4 – Completely Static POR of Probabilistic Systems	29
4.1. Static POR for Non-Probabilistic Systems	29
4.2. Static POR for Probabilistic Systems	32
4.2.1. Fundamental Considerations	33
4.2.2. Preliminary Calculations	34
4.2.3. Naïve SPOR	35
4.2.4. Reachability-Aware SPOR	37
4.2.5. Experiments	51
<hr/>	
CHAPTER 5 – Conclusion and Future Lines of Work	59

1

Introduction

The presence of software systems in everyday life has exponentially increased during the last decades. Their use ranges from the most trivial tasks, like driving and controlling domestic appliances, to more sophisticated jobs, which include air traffic synchronization, logistics or data management, among others. It is very likely also that the reader has required the use of some software entities in order to read the present document, e.g. obtaining it from the Internet or displaying it in a computer screen, smart-phone or electronic-book reader. This quotidian use of software requires the development of reliable programs. Nevertheless, it is well known that software systems quite often contain errors, some of them very subtle and hard to identify. As evidence of the high relevance of error identification in the software development process, we can recall that, as stated in [Eme08], software developers usually devote half of their time to error identification and system debugging. There exist several techniques whose goal is to analyze whether a system provides the expected functionality, which corresponds to a system specification. That analysis process is known as *Verification*.

The different existing techniques for Software Verification can be classified attending to the way they are performed and, consequently, the kind of information they collect. *Dynamic Verification* techniques are performed during system execution, thus requiring its prior total or partial implementation. These techniques usually analyze the behaviour of the system through its output to some controlled stimuli. One very popular dynamic verification technique is known as testing, where the specification of the system is given in the shape of a series of tests, named test suite, that the system has to pass in order to be considered correct. Conversely, *Static Verification* techniques physically analyze the system implementation or a model of it. Among these techniques we can find several approaches that vary from system metrics calculation, e.g. lines of code or size of binaries, to common bad practices identification or the use of mathematics in order to prove the correspondence between the system specification and its implementation. The use of these latter approaches is known as *Formal Verification*, which utilizes a set of mathematically-based techniques known as *Formal Methods*.

Formal verification techniques require the specification of the system in a mathematical and logical way. That specification is then checked against the code implementation in order to identify whether the latter meets the former. The most popular formal verification techniques are *Logical Inference* and *Model Checking*. Logical inference, commonly referred also as theorem proving, evaluates the correctness of the implementation with regards to its implementation by using certain axiomatic system and a set of inference rules. This task can be automatically or semi-automatically performed by a tool known as theorem prover. Well-known examples of these tools are Isabelle [Pau94] and Coq [HKPM97]. On the other hand, model checking techniques evaluate the correctness of the implementation with the use of a model of that implementation, represented in a Kripke Structure, and a set of properties, specified in some temporal logic formula, that conform the system specification. The aim of this technique is determining whether the model provided correctly models the logical function. Some examples of successful model checking tools, known as model checkers, are Prism [KNP02] and SPIN [Hol03].

The use of model checking techniques is specially recommended for the formal verification of concurrent systems. This kind of systems are composed by a series of processes that interact with each other either via an explicit communication mechanism or by sharing resources. One of the main characteristics of these systems is their non-determinism. We refer to a system as non-deterministic when it can provide different outputs from the same set of inputs. That non-determinism typically derives from the lack of a global scheduler that determines an execution order for the processes with respect to each other. Therefore, the different possible interleavings of process actions may result in a different output of the system. Apart from non-determinism, there exist some systems that show some uncertainty in the effects of their actions. Such uncertainty can derive, for instance, from the existence of randomized actions, whose effect is chosen stochastically with respect to some distribution, or the presence of faulty communication channels between processes. The systems that exhibit this kind of uncertainty modelled by stochastic actions are referred as *probabilistic systems*.

Regarding the way in which the model of the system being verified is represented, there exist two alternatives. The first implies using a modelling language, like PROMELA [PRO] which is used as input language for the popular SPIN model checker, to build the system. The second alternative allows the use of the source code as model of the system being verified, as occurs in the model checker McErlang [FS07].

Concerning the properties that conform the specification, they are most commonly specified in a temporal logic or an equivalent structure, like regular automata. Temporal logics allow the statement of properties that permit the reasoning about the behaviour of concurrent programs taking into account a temporal component. For instance, they allow the enunciation of propositions that can eventually hold in a future state of the system with respect so some other moment in time. Probably the most popular temporal logics are a linear time one known as Linear Temporal Logic (LTL), see [Pnu77], and a branching time logic named Computation Tree Logic (CTL), described in [CE82] and studied in [EH82]. In the context of linear time logics, from each moment in time, thus from each state of the system, there exists only one possible future, whereas the semantics of branching time logics allow the existence of several possible futures, which can also be quantified. However, as CTL and LTL are not equivalent, which was shown in [Var98], the superset CTL* of both of them was generated and also described in [Var98]. All these logics allow the construction of *qualitative properties* of the system, e.g. in a communication system one could try to check whether the property “when a message is sent, there exist some execution paths in which the message is correctly delivered” holds or not. Nevertheless, in the real world exists the necessity to model some uncertainty about the system, as we stated before, specially when trying to analyze some aspects of a system, like the so-called Quality of Service. Therefore, the temporal logics were extended with a component known as “probabilistic operator”, which allows the construction of predicates that refer to the probability to satisfy certain predicate or propositional formula. Then, some probabilistic temporal logics like PCTL and its generalization PCTL*, see e.g. [HJ94], [Din07], were described in order to allow the statement of *quantitative properties*.

As we mention before, the goal of a model checking process is, given a model M of the system and a property P , identify whether M is a model for P if the system starts from one of the possible initial states s of M . In order to do that, the most common approach consists in, first, generating the automaton for property $\neg P$, named $A_{\neg P}$. Then, the crossproduct $M \otimes A_{\neg P}$, from initial state s , is generated and checked for acceptance-emptiness. If there is no word w that is accepted by the crossproduct, then $M, s \models P$, which means that the property holds for the system. Otherwise, if w is accepted, the property does not hold and w represents a counterexample for P . Therefore, the model checking process can be considered as a reachability analysis one. However, this process suffers the drawback of generating a number of states, known as *State Space*, that grows exponentially with respect to some features, like the number of processes involved in the system or the complexity of the property. That exponential growth in the state space can make the model checking process impossible in the real-time setting due to both storage and computation restrictions.

This problem is known as the *State Space Explosion Problem*.

In order to tackle the state space explosion problem, there exist several approaches that perform model checking avoiding the generation of the complete state space of a system. One of them is known as *Abstraction*, where some details of the system are ignored in order to generate a smaller model \bar{M} . That reduced model \bar{M} should be easier to analyze than M and the checks over \bar{M} should be sufficient to reason about M . For instance, an exact abstraction guarantees that $\bar{M} \models P \Leftrightarrow M \models P$. Nevertheless, there are some more conservative abstractions that can only ensure $\bar{M} \models P \Rightarrow M \models P$. Another alternative consists in the symbolic representation of sets of states and transitions instead of representing all of them individually. This approach is known as *Symbolic Model Checking*, see [BCM⁺90], in contraposition to *Explicit Model Checking*. Symbolic Model Checking makes use of some structures for the representation of relations and states as formulas, known as Ordered Binary Decision Diagrams (BDDs). BDDs were introduced in [Bry86], and have been generalized in structures like the Multi-Terminal Binary Decision Diagrams, from [FMY97], which were successfully used to perform symbolic model checking over a probabilistic system, as described in [BCHG⁺97]. A different technique for the reduction of the state space is known as partial order reduction. This technique uses information about the different actions that each process can perform in order to, together with a notion for action dependence, establish which of the possible actions from every state s can be safely delayed to generate a reduced, but equivalent, state space. The sequences of actions that are represented in the reduced state space are referred to as *representatives*. Attending to what kind of information these techniques collect, with respect to temporal criteria, they can be classified into *stubborn/persistent sets* or *sleep sets*. Sleep sets, introduced in [God96], try to identify the dependencies between actions by analyzing only the actions enabled from a state s and the actions that have already been executed. Conversely, stubborn sets, which are described in [Val89], use information about the future executable actions. A very popular stubborn set technique is known as *ample set* approach, in which the persistent set, referred to as ample set, has to fulfill a series of conditions, described in [Pel93]. Depending on the moment in which these techniques obtain the information and decide on the different representatives, partial order reductions are referred either as *static* or *dynamic*. Static partial order reductions decide on the representatives by performing a static analysis over the structure of the system being analyzed. In order to do that, they need to generate a symbolic representation of the system and generate the conditions to only construct those representatives by, for instance, modifying the program semantics of each individual process in a conservative way. Conversely, dynamic partial order reductions discover the representatives during the model checking process, i.e. the reachability analysis, of the property.

Contribution of the Thesis The contribution of this master thesis work to the state-of-the-art of model checking is twofold. On the one hand, we develop two new techniques for the static partial order reduction (SPOR) of software systems. Both techniques are developed with the aim of being suitable for both probabilistic and non-probabilistic systems, hence allowing the analysis of qualitative and quantitative properties. The two of them can be applied over the probabilistic control graphs of the processes that compose a system and generate a modified version of them such that the state space of the new system either has the same size or is smaller than the state space of the original unreduced one. The use of each of our SPOR techniques is determined by the temporal logic used to build the properties checked. The first of the techniques, to which we refer as *Naïve SPOR*, can be applied for the sound reduction of the state space of a system where the properties to be checked can be expressed in PCTL* and are stutter-invariant, notion that is later explained in this document. The second technique, named *Reachability-Aware SPOR*, can only be applied for the reduction of models where the properties checked, both qualitative and quantitative ones, are stutter-invariant and can be also expressed in linear time logic. Nevertheless, we show evidence of how *Reachability-Aware SPOR* can be more efficient than *Naïve SPOR*, in terms of providing a higher reduction to the same state space.

On the other hand, we perform an extension of the model checker LiQuor [CC06]. This model checker performs explicit model checking for both qualitative and quantitative linear time properties of systems. The input language for the models that this model checker accepts is PROBMELA, see [BCG04]. PROBMELA programs are translated into an XML-like intermediate language named PASM. Besides, LiQuor is able to apply dynamic partial order reductions to PASM programs. The contribution of our extension comes, again, in two ways:

- We applied a reverse engineering process in order to gather all information contained in PASM programs sufficient to build a symbolic representation of the system. That symbolic representation of the system is composed by all different probabilistic control graphs of the processes and is now built and stored by the model checker. Therefore, some state space reduction techniques can be applied to this representation, fact that represents an increase in the extensibility of LiQuor functionality.

- Given the symbolic representation of the system, we include the functionality to perform both *Naïve SPOR* and *Reachability-Aware SPOR* techniques. The way in which this implementation acts implies analyzing the symbolic representation and modifying PASM code. Therefore, as the modification is performed in the intermediate code and not in the source code, those techniques could be applied to any input language that could be compiled into PASM.

Structure of the thesis. In Chapter 2 of this master thesis we elaborate on the basic notations and provide with the knowledge required for the perfect understanding of the contents presented. Moreover, in that chapter we also introduce the state space explosion problem along with a description of partial order reduction techniques based on ample set identification, as motivation for the present thesis work. In Chapter 3 we describe a technique that allows a semi-static partial order reduction, as allows the identification of some actions that, when included in an ample set, guarantee that some of the ample set conditions are fulfilled. This topic composes a whole topic on its own because it derives from work previous to this thesis and has not been mentioned in the contributions. Nevertheless, it is included in present document because it represents the preliminary work required as preparation for the thesis. In Chapter 4, we introduce *Naïve SPOR* and *Reachability-Aware SPOR* techniques, as well as an already existing technique that served as basis for them. Besides, we illustrate both techniques and compare their use in combination with symbolic and explicit model checking. We present some practical examples, provide formal proof of the soundness of the new techniques and analyze their performance based on a comparison over empirical results. Finally, in Chapter 5 we present the conclusion and future research lines that could be taken as continuation of this thesis work.

2

Basic Concepts

Before we start explaining our contribution to Model Checking, we give a brief introduction to the notations that will be necessary to understand the context of the present thesis document. Besides, we elaborate on one of the main difficulties this formal verification technique faces, as well as a short description of one of the already existing techniques to alleviate it.

2.1. Notations

Model Checking technique typically requires the generation of a model of the system, which would be later algorithmically examined. One popular model representation mechanism for probabilistic systems are Markov Decision Processes (MDP for short), which reflect some operational behaviour of the system under consideration. A Markov Decision Process M is defined by a tuple $(S, Act, \Longrightarrow, s_0, AP, L)$ where:

- S is a set of states
- Act is a set of Actions
- $\Longrightarrow: S \times Act \mapsto Distr(S)$ is the partial transition function. $Distr(S)$ is the set of all stochastic distributions over countable set S . Therefore, for every state $s \in S$ and action $\alpha \in Act$ for which the function \Longrightarrow is defined, we obtain a stochastic distribution $\mu \in Distr(S)$ s.t. for every state $s' \in S$. $\mu(s') \in [0, 1]$ and $\sum_{s' \in S} \mu(s') = 1$
- $s_0 \in S$ is the initial state
- AP is a set of atomic propositions
- $L: S \rightarrow 2^{AP}$ is the labeling function

This kind of model represents the behavior of the whole system by including all the possible states in which the system can appear, along with the transitions between them and the atomic propositions for their valuations. Intuitively, the system modeled starts in initial state s_0 , and evolves to different states according to the transition relation \Longrightarrow . In the case

of software systems, each different state $s \in S$ holds information about the values of all variables along with the program counters that identify the execution point of each of the programs that compose the system. The set S of all states is frequently referred to as the *State Space* of a system. An MDP is said to be finite if its number of states and transitions is not infinite.

This model representation allows the construction of a directed labelled graph with the aim of easing the overall understanding of the system. In this graph, each node corresponds to a different state s of the state space. Besides, for every state-action pair $(s_i, \alpha) \in S \times Act$ for which \Longrightarrow is defined with output μ and for every $s_j \in \text{support}(\mu) = \{s_k \in S \mid \mu(s_k) > 0\}$, there is a different edge between graph nodes n_i and n_j where n_i and n_j correspond to s_i and s_j respectively. In this document, we will represent those edges labeled as $\alpha[\mu(s_j)]$ or only α if $\mu(s_j) = 1$. Notice that if $s_i = s_j$, the edge is a self loop.

As a matter of example, consider the system, to which we refer as *the stochastic counter*, composed by a program which operates over an integer variable x bounded to interval $[0, 2]$ and a binary value a . This system increments variable x by one unit iteratively until it reaches value 3. Then, it stochastically changes value of a to *true* or *false*, with the same probability for both alternatives. The MDP that models such program can be $M = (S, Act, \Longrightarrow, I, AP, L)$ where:

- $S = \{s_i\}$ where $i \in [0, 7]$
- $Act = \{\alpha, \beta, \gamma\}$, where α increments value of x in one unit, β stands for the action that probabilistically sets variable a to either value *true* or *false* and γ assigns value 0 to x .
- \Longrightarrow is defined for cases:
 - $\Rightarrow(s_0, \alpha) = \mu_{0, \alpha}$ s.t. $\mu_{0, \alpha}(s_1) = 1$ and $\mu_{0, \alpha}(s_n) = 0$ for every $s_n \in S \setminus \{s_1\}$
 - $\Rightarrow(s_1, \alpha) = \mu_{1, \alpha}$ s.t. $\mu_{1, \alpha}(s_2) = 1$ and $\mu_{1, \alpha}(s_n) = 0$ for every $s_n \in S \setminus \{s_2\}$
 - $\Rightarrow(s_2, \gamma) = \mu_{2, \gamma}$ s.t. $\mu_{2, \gamma}(s_3) = 1$ and $\mu_{2, \gamma}(s_n) = 0$ for every $s_n \in S \setminus \{s_3\}$
 - $\Rightarrow(s_3, \beta) = \mu_{3, \beta}$ s.t. $\mu_{3, \beta}(s_0) = 0.5$, $\mu_{3, \beta}(s_4) = 0.5$ and $\mu_{3, \beta}(s_n) = 0$ for every $s_n \in S \setminus \{s_0, s_4\}$
 - $\Rightarrow(s_4, \alpha) = \mu_{4, \alpha}$ s.t. $\mu_{4, \alpha}(s_5) = 1$ and $\mu_{4, \alpha}(s_n) = 0$ for every $s_n \in S \setminus \{s_5\}$
 - $\Rightarrow(s_5, \alpha) = \mu_{5, \alpha}$ s.t. $\mu_{5, \alpha}(s_6) = 1$ and $\mu_{5, \alpha}(s_n) = 0$ for every $s_n \in S \setminus \{s_6\}$
 - $\Rightarrow(s_6, \gamma) = \mu_{6, \gamma}$ s.t. $\mu_{6, \gamma}(s_7) = 1$ and $\mu_{6, \gamma}(s_n) = 0$ for every $s_n \in S \setminus \{s_7\}$

– $\Rightarrow (s_7, \beta) = \mu_{7, \beta}$ s.t. $\mu_{7, \beta}(s_0) = 0.5$, $\mu_{7, \beta}(s_4) = 0.5$ and $\mu_{7, \alpha}(s_n) = 0$ for every $s_n \in S \setminus \{s_0, s_4\}$

- s_0
- $AP = \{0, 1, 2, 3, a\}$, which reflects the numerical and truth values of variables x and a respectively.
- L such that :
 1. $L(s_0) = \{0, a\}$
 2. $L(s_1) = \{1, a\}$
 3. $L(s_2) = \{2, a\}$
 4. $L(s_3) = \{0, a\}$
 5. $L(s_4) = \{0\}$
 6. $L(s_5) = \{1\}$
 7. $L(s_6) = \{2\}$
 8. $L(s_7) = \{0\}$

The directed graph that corresponds to the *stochastic counter* is presented in Figure 2.1.

As we can see, the MDP provides the intuitions of which states or situations are reachable (i.e. possible) within a program execution. However, as the choice between the transitions is performed non-deterministically, in more complex systems, for instance involving different concurrent processes, it is not possible to reason, a priori, about the individual behaviour of each program. For instance, in the MDP represented in Figure 2.1 the conditions that enable each transition are abstracted and omitted. Therefore, the set of relevant properties of the system to be verified is shrunk. In order to provide a deeper insight into the programs, a closely related representation construct, known as *Probabilistic Control Graph* is used. A probabilistic control graph is a tuple $PCG = (Loc, Event, \rightsquigarrow, l_0)$ over a set Var of variables, and channels, where:

- Loc is a set of locations. A location determines a position into the program, but abstracts the value of any variable.
- $Event$ is a set of process events.
- $\rightsquigarrow \subseteq Loc \times Cond(Var) \times Distr(Event \times Loc)$ is the edge relation. $Cond(Var)$ is the set of conditional propositions that can be elaborated with the variables contained

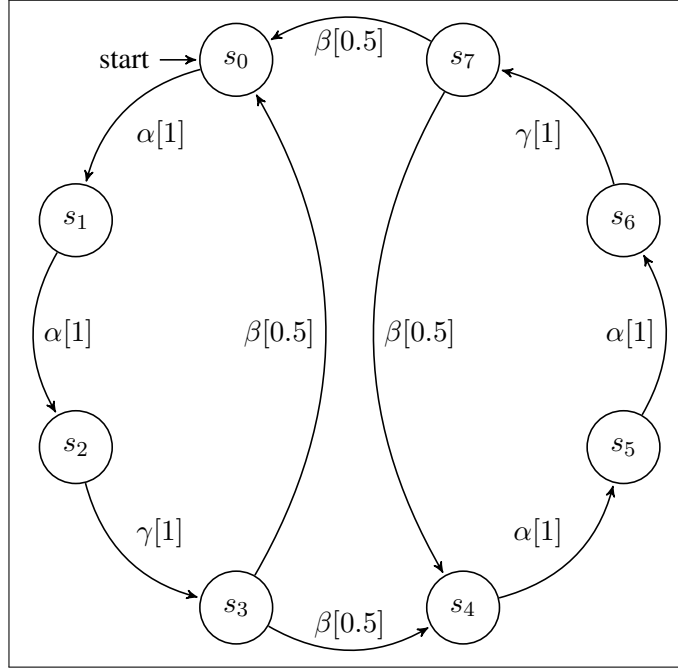


Figure 2.1: MDP digraph for the stochastic counter

in Var . For instance, $x > 1$ for a numerical variable x or $y = true$ for a boolean variable y . $Distr(Event \times Loc)$ is the set of all possible distributions over pairs $(e, l) \in Event \times Loc$.

- l_0 is the initial location.

Besides, we can define the set of actions $Act \subseteq Cond(Var) \times Distr(Event \times Loc)$ as $Act = \{\alpha \in Cond(Var) \times Distr(Event \times Loc) \mid \alpha \in \bigcup_{l \in Loc} \rightsquigarrow(l)\}$. For the sake of simplicity, from now on, each tuple $(l, g, Distr) \in \rightsquigarrow$, which corresponds to an action α will be depicted as $l \rightsquigarrow^{g:\alpha} Distr$. $g \in Cond(Var)$ is called guard and represents a condition that has to be fulfilled for the action α to be enabled, which means that the effect of each event $e \in Event$ can be applied regarding probability distribution $Distr$. Therefore, for the example in Figure 2.1. the probabilistic control graph $PCG = (Loc, Event, \rightsquigarrow, l_0)$ over $Var = \{x\}$ could be:

- $Loc = \{l_0, l_1\}$. Usually, the locations of a program correspond to positions from which the program can execute at least one action.
- $Event = \{\alpha, \beta_1, \beta_2, \gamma\}$. Which correspond to the events associated to actions described in Figure 2.1, whose effect over a valuation $\eta \in Val(Var)$ is:

1. $Effect(\alpha, \eta) = \eta_{[x=x+1]}$

2. $\text{Effect}(\beta_1, \eta) = \eta_{[a=true]}$
 3. $\text{Effect}(\beta_2, \eta) = \eta_{[a=false]}$
 4. $\text{Effect}(\gamma, \eta) = \eta_{[x=0]}$
- $\rightsquigarrow = \{(l_1, g_1 = (0 < x < 2), Distr_1), (l_1, g_2 = (x = 2), Distr_2), (l_1, g_3 = true, Distr_3)\}$
where:
 - $Distr_1(\alpha, l_1) = 1$
 - $Distr_2(\gamma, l_2) = 1$
 - $Distr_3(\beta_1, l_1) = 0.5, Distr_3(\beta_2, l_1) = 0.5$

We can notice that, from \rightsquigarrow , we can extract the set of actions $Act = \{\alpha, \beta, \gamma\}$.

- $l_0 = l_1$

In a similar way to the process of building a digraph of a MDP, it is possible to build one that represents a probabilistic control graph. The digraph in Figure 2.2. corresponds to this later probabilistic control graph. It is also possible to obtain the MDP that corresponds to the set of probabilistic control graphs of the different processes that compose a system. Such procedure is explained in [Cie11].

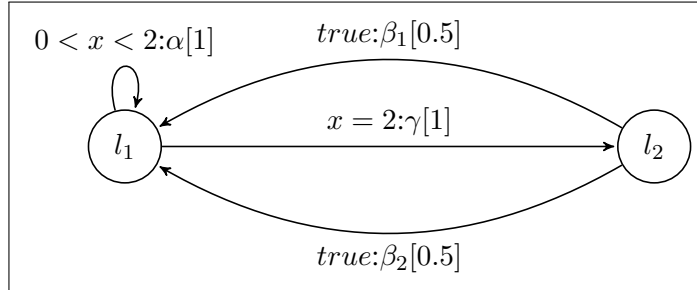


Figure 2.2: Probabilistic Control Graph for the stochastic counter

2.2. State Space Explosion Problem

Nowadays, most software applications are no longer composed by a single execution thread, but as a set of processes running simultaneously and concurrently. The formal verification of this kind of multiprocessed systems is a quite computationally expensive task. Model Checking techniques offer some ways for carrying out such verification tasks. As we mentioned before, there exist some algorithms to unfold a probabilistic control graph PCG_i , or a set of them, into a MDP, e.g. in [Cie11], $MDP_i = MDP(PCG_i)$. This technique is

especially useful when we try to build the MDP that corresponds to a system composed of several processes. In order to do that, first we must obtain the probabilistic control graphs of each different process. Then, an interleaving operation has to be performed between them in order to obtain their crossproduct. Finally, this crossproduct can be unfolded into a MDP.

In order to provide a better understanding of this process, we show an example of this procedure. Consider a system, which we name the *two-counters system*, composed by two programs P_1 and P_2 . These programs perform the behavior of a *stochastic counter* over integer variables x, y and binary variables a, b respectively. The probabilistic control graphs PCG_1 and PCG_2 corresponding to them are presented in Figure 2.3. Once we have them, the crossproduct is built by first generating a control state that corresponds to the initial state of both programs. Then, a new state is generated for every single transition with probability greater than 0 that each program can take. As variable values are abstracted, only the locations of each probabilistic control graphs are taken into account in order to differentiate crossproduct locations. Therefore, a crossproduct location which corresponds to the same single program locations is unique. The crossproduct probabilistic control graph for the *two-counters system*, over the variable set $Var_{1,2} = Var_1 \cup Var_2$, is shown in Figure 2.4. As we can see, the set of locations of the crossproduct is $Loc_{1,2} = Loc_1 \times Loc_2$.

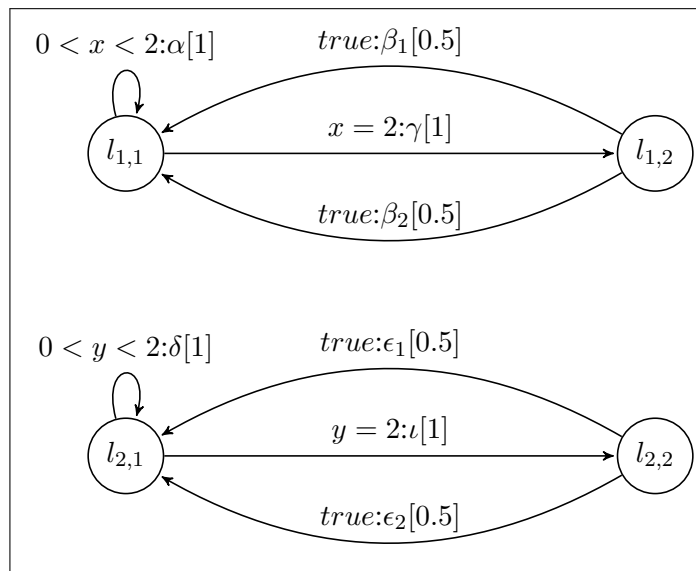


Figure 2.3: Individual Probabilistic Control Graphs for the 2-counters system

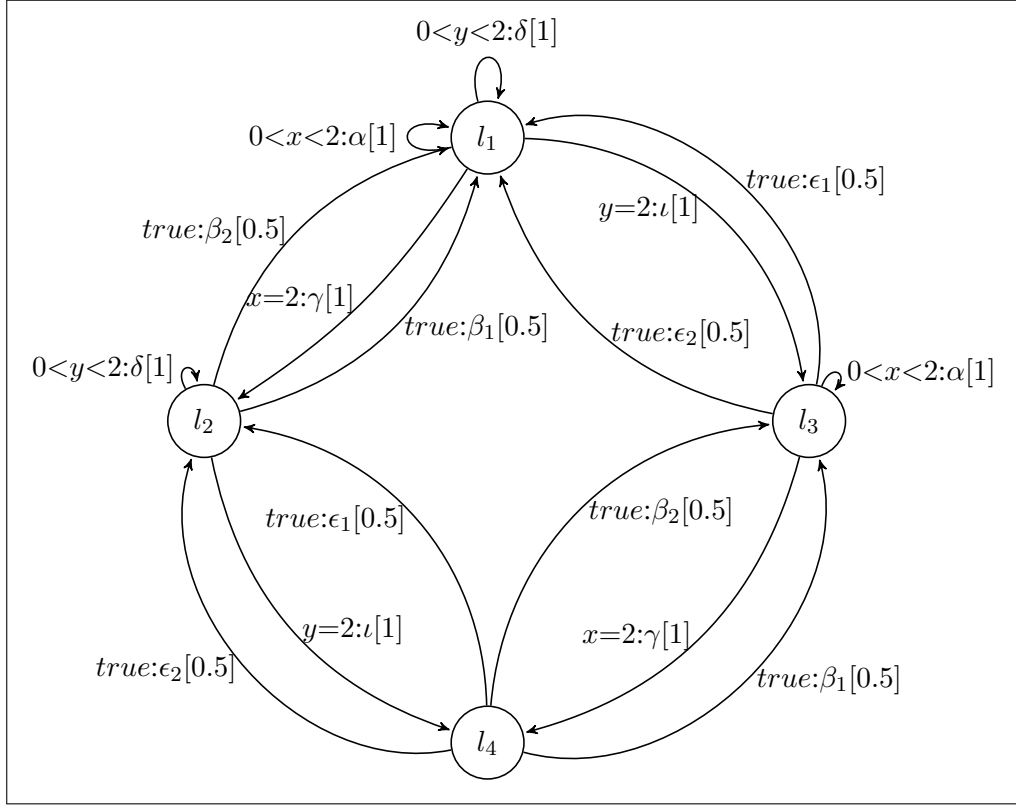


Figure 2.4: Crossproduct Probabilistic Control Graph for the 2-counters system

Analogously, in the case of a system composed of n processes, the crossproduct probabilistic control graph obtained from the interleaving of all them contains a location set $Loc_{1-n} \subseteq Loc_1 \times \dots \times Loc_n$ (notice we use \subseteq because of the possible presence of synchronous communication operations [BK08], if they are not present, “=” could be used). Therefore, the number of locations in a crossproduct probabilistic control graph can increase exponentially with respect to the number of processes it was calculated from, as $|Loc_{1-n}| \approx \prod_{i=1}^n |Loc_i|$.

Regarding the cardinality of the state space S_i of a probabilistic control graph PCG_i over variable set Var_i , it can be at most $|Loc_i| \cdot \prod_{v \in Var_i} |dom(v)|$. Therefore, in the case of the *two-counters system*, where $dom(x) = dom(y) = 0, 1, 2$ and $dom(a) = dom(b) = 0, 1$, $|S_{1,2}| = |Loc_1| \cdot |Loc_2| \cdot |dom(x)| \cdot |dom(y)| \cdot |dom(a)| \cdot |dom(b)| = 2^2 \cdot 3^2 \cdot 2^2 = 144$. Then, we can see how the cardinality of the state space increases exponentially in both the number of processes it is composed of and the number of observable variables. Because of that, even for simple and small systems, the number of states can be too huge. For instance, a

system composed of 6 processes with 10 locations each, over a total set of 8 integer variables which range between 10 possible values, generates a state space S with cardinality $|S| = 10^6 \cdot 10^8 = 10^{14}$ states! This phenomenon is known as *State Space Explosion* and represents the major problem of Model Checking, as stated in [Cla08], due to its enormous increase in the complexity to analyse relatively big systems, also making their verification intractable. In the next section, we describe one technique used to ease this problem.

2.3. Partial Order Reductions

This technique aims at identifying a smaller reduced model $\hat{M} = (\hat{S} \subset S, Event, \implies, s_0, AP, L)$ of the state space of a system represented by the MDP M such that, given Φ as the set of all stutter-invariant properties it holds that $\forall \phi \in \Phi. M \models \phi \Leftrightarrow \hat{M} \models \phi$. In that case we say that M and \hat{M} are stutter-trace equivalent. Instances of stutter-invariant properties are those specified in a temporal logic like LTL_{\circ} , which are LTL formulas that do not contain *next* operator \circ . Those properties do not differentiate traces which are equal if we eliminate the consecutive states where the valuation of the atomic propositions remains unchanged. Then, the properties of the unreduced system M can be checked by analysing the reduced system \hat{M} . \hat{M} is obtained by removing some actions from the set of enabled ones in certain state $s \in S$, consequently, removing some paths and states from the unreduced system. From now on, we refer to the set of all the enabled actions from a state $s \in S$ as $Act(s)$.

As sample introduction to why POR are possible, consider a small subset of the state space of the *two-counters system*, shown in Figure 2.5. Here, every state is labelled as s_{xyab} in order to show the value of variables x, y, a and b , e.g. s_{01tf} represents a state where $\eta(x) = 0, \eta(y) = 1, \eta(a) = true$ and $\eta(b) = false$. This subset originates from all the possible interleavings of actions $\alpha, \beta, \gamma, \delta, \epsilon, \iota \in Act$ from state s_{00tt} . If we wanted to check a system property which only refers to variables a and b , therefore $AP = \{a, b\}$, it may suffice to analyze only some interleavings, like the ones in Figure 2.6, as original unreduced state space could be built varying the execution order of actions that do not modify the variables in AP . We can see how the new interleaving does not disable actions γ and ι , which is a relevant property on which we elaborate later in this section.

In order to determine the interleavings that have to be performed, we concentrate in the technique called *ample set method*, described in [Pel93]. This technique aims at identifying the ample set: $ample(s) \subseteq Act(s)$. Then, the interleaving is only performed for the actions in $ample(s)$ and the rest of enabled actions are delayed and executed later from some of the successor states of s . The execution sequences that are contained in the reduced state space

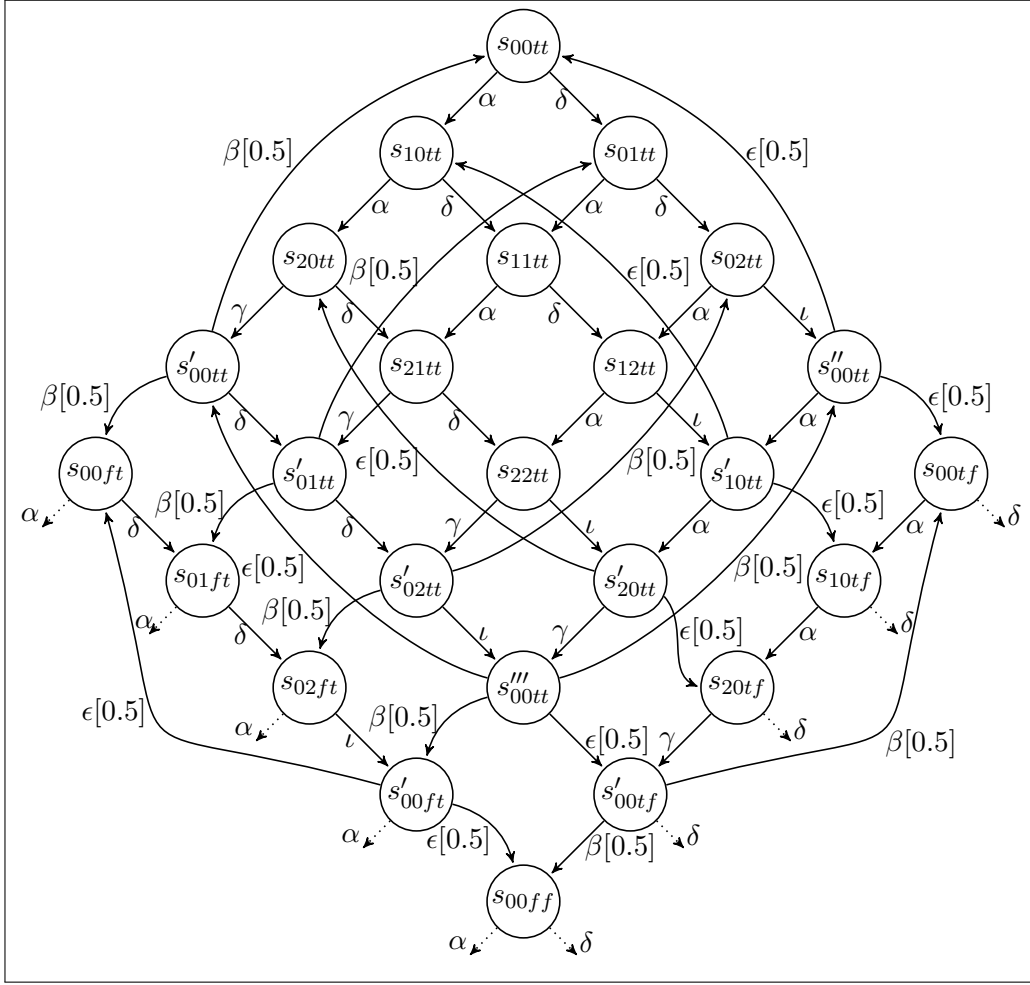


Figure 2.5: Fully interleaved subgraph of the 2-counters system

\hat{S} are named representatives, as they will represent every other possible execution sequence in the unreduced state space S . Before we describe the conditions that ensure the selection of certain ample set preserves system properties, we provide some definitions required to understand the aforementioned properties:

- S' : is the reduced state space $S' \subseteq S$.
- $s = \langle l_0, \dots, l_n, \eta \rangle$: is a state of the state space S . It is composed by the control locations $l_i \in Loc_i$, where $1 \leq i \leq n$ of each of the n processes in the system, along with the valuation η of the set Var of variables and $Chan$ of channels.
- $enabled(l_i) \in Act(s)$: is the set of all actions of state s that belong to location l_i from PCG_i .

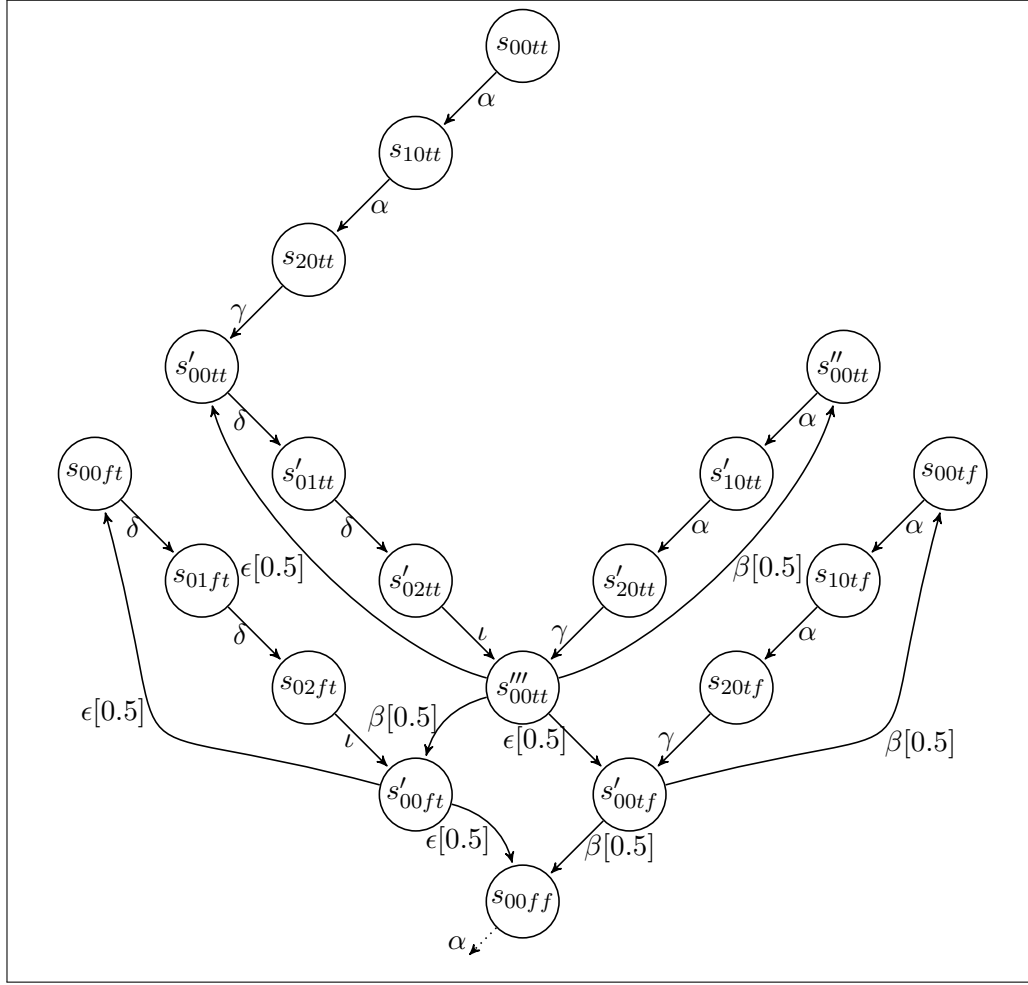


Figure 2.6: Single interleaving subgraph of the 2-counters system

- **α -successor:** we refer as such to those states $s' \in S$ for which $\Longrightarrow(s, \alpha) = \mu_\alpha$ where $\alpha \in Act(s)$ and $\mu_\alpha(s') > 0$. We depict the set of α -successors of s as $\alpha(s)$.
- **Transition Probability:** as mentioned before, in probabilistic systems when an action $\alpha \in Act$ is performed from state $s \in S$, the successor state $t \in \alpha(s)$ is determined probabilistically. We refer to that probability as $P(s \Rightarrow^\alpha t) = \mu_\alpha(t)$, where $\Longrightarrow(s, \alpha) = \mu_\alpha$.
- **Action Independence:** we say two different actions $\alpha, \beta \in Act$ are independent in a MDP if and only if for all states $s, t, u \in S$ where $\alpha, \beta \in Act(s)$ it holds that:
 1. $P(s \Rightarrow^\alpha t) > 0$ implies $\beta \in Act(t)$
 2. $P(s \Rightarrow^\beta t) > 0$ implies $\alpha \in Act(t)$
 3. for all states $w \in S$: $\sum_{t \in S} P(s \Rightarrow^\alpha t) \cdot P(t \Rightarrow^\beta w) = P(s \Rightarrow^\beta u) \cdot P(u \Rightarrow^\alpha w)$

If two actions are not independent, they are dependent.

- **Action Visibility:** $\alpha \in Act$ is called invisible, or stutter, if $\forall s \in S, \alpha \in Act(s), s' \in \alpha(s). L(s) = L(s')$. Otherwise, α is visible.
- **Path:** it is a sequence $\rho = s_0 \Rightarrow^{\alpha_1} s_1 \dots \Rightarrow^{\alpha_n} s_n$ such that $P(s_{i-1} \Rightarrow^{\alpha_i} s_i) > 0$ for $1 \leq i \leq n$.
- **Cycle:** a cycle is a path $\rho = s_0 \Rightarrow^{\alpha_1} s_1 \dots \Rightarrow^{\alpha_n} s_n$ such that $s_0 = s_n$.
- **Full Expansion:** a state $s \in S$ is fully expanded if $ample(s) = Act(s)$.
- **Subgraph:** a subgraph of MDP M is a tuple $\xi = (T, Act_\xi)$ where:
 - , e.g. properties specified in LTL which do not contain *next* operator \circ ,
 - 1. $\emptyset \neq T \subseteq S$.
 - 2. $Act_\xi : T \rightarrow 2^{Act}$ such that for all $t \in T$ we have $\emptyset \neq Act_\xi(t) \subseteq Act(t)$.
 - 3. If $t \in T, \alpha \in Act_\xi$ and $P(t \Rightarrow^\alpha u) > 0$ for $u \in S$, then $u \in T$.
- **End Component:** an end component is a subgraph whose underlying digraph $G_\xi = (T, E)$ with $(s, s') \in E \Leftrightarrow \exists \alpha \in Act_\xi$ such that $P(s \Rightarrow^\alpha s') > 0$ is strongly connected.

In order to guarantee that interleaving only the actions in the ample set does not change the properties that the reduced system models, this set has to fulfill a set of conditions, already described, for instance, in [Pel98], [Cie11], [CGC04]:

C0: $ample(s) = \emptyset \Leftrightarrow Act(s) = \emptyset$

C1: for every finite path $\rho = s \Rightarrow^{\alpha_1} \dots \Rightarrow^{\alpha_n} s_n \Rightarrow^\beta s_{n+1}$ in M , if β depends on $ample(s)$ then exists at least one α_i for $i \in [1, n]$ s.t. $\alpha_i \in ample(s)$

C2: if s is not fully expanded then all the transitions in $ample(s)$ are invisible.

C3: for each end component (T, A) in \hat{M} : $\alpha \in \bigcap_{t \in T} Act(t)$ implies $\alpha \in \bigcup_{t \in T} ample(t)$

This four conditions are sufficient for non-probabilistic systems. However, in order to check quantitative properties in the probabilistic setting, they are not enough, as shown in e.g. [CGC04] and [DN04]. Therefore, [GKPP99] introduces the *Branching Condition* which is discussed in [DN04] and [BDG06] for its necessity in probabilistic systems:

C4: if $\text{ample}(s) \neq \text{Act}(s)$ then $|\text{ample}(s)| = 1$.

This condition ensures stutter-trace equivalence between M and \hat{M} for stutter-invariant qualitative or quantitative properties expressed in branching or linear temporal logics, expressible in $PCTL^*_{-\circ}$, as stated in [BDG06]. However, in [CGC04] a weaker condition is presented which is valid for the reduction of systems whose qualitative and quantitative properties are expressed in $LTL_{-\circ}$:

C4': if $\rho = s \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} s_n \xrightarrow{\beta} s_{n+1} \dots$ is a path in M where $s \in S$, $\alpha_1 \dots \alpha_n, \beta \notin \text{ample}(s)$ and β is probabilistic then $|\text{ample}(s)| = 1$.

The informal meaning of condition C0 refers to the prohibition to include final states, i.e. states where no more transitions can be taken, in \hat{M} which were not already present in M . Regarding C1, it guarantees that every finite path in M starting on certain state s , all the actions $\gamma \in \text{Act}(s)$ and $\gamma \notin \text{ample}(s)$ are independent from the actions in $\text{ample}(s)$, as an action α dependent on $\text{ample}(s)$ cannot appear by executing only γ . Condition C2 guarantees that executing the actions in $\text{ample}(s)$ generates stutter-equivalent execution traces, i.e. execution traces in which the valuation of the atomic propositions does not change. Condition C3 avoids the removal of an action as side-effect of a loop over invisible actions that belong to the same end component. Finally, conditions C4 and C4' guarantee that path probabilities are maintained. Then, as proven in [BDG06] if \hat{M} is built using the ample set method where conditions C1-C4 hold, $\forall \phi \in \Phi$ such that ϕ is a stutter-invariant temporal qualitative or quantitative property expressed in $PCTL^*_{-\circ}$ (which includes every property expressible in $LTL_{-\circ}$ or $CTL_{-\circ}$). $M \models \phi \Leftrightarrow \hat{M} \models \phi$ also holds. Analogously, if C1-C3 and condition C4', which is weaker than C4, hold, stutter trace equivalence for stutter-invariant qualitative and quantitative linear time properties is also guaranteed, as proven in [CGC04].

2.3.1. Some POR Alternatives

There exist different alternatives for the generation of the reduced state space \hat{S} . Regarding the moment in which the different processes of the system are analyzed, in order to check the fulfilment of the POR conditions for the ample sets, they can be categorized in two main groups:

Dynamic Partial Order Reductions. In this category, the checking of the fulfilment of at least one of the ample set conditions is performed on an on-the-fly basis, that is to say, while the model of the global system is being generated. There exist many different algorithms

that allow the successful implementation of these techniques. One of them is presented in [FG05]. There are some other similar implementations, like the one in [Cie11], which provides some overapproximations sufficient to guarantee the conditions of the ample sets identified. Nevertheless, some of these approximations strongly depend on the generation of the state space via a Depth First Search. Otherwise, the computational complexity of the algorithm to guarantee the ample set conditions would raise significantly.

Static Partial Order Reductions. Opposite to dynamic partial order reductions, all the condition checking and reductions are performed prior to state space generation. In order to do that, the system has to be symbolically represented in a smaller structure. That structure would frequently be composed by the probabilistic control graphs (recall that a program graph can be considered as a probabilistic control graph where each stochastic distribution enables at most one transition and that transition is taken with probability one) of the processes that integrate the system. Then, all the necessary checks to identify the representatives are performed on the different control locations and transitions of the probabilistic control graphs. Possible implementations, for non-probabilistic systems, are described in [KLM⁺98] and [BK08]. They describe a way of identifying which transitions should be performed by analyzing the program graphs of the individual processes, that is to say, prior to the global state space generation. One big advantage of these techniques is that, as all reduction is decided prior to reachability check, they are independent to the search algorithm used. Moreover, they can be easily combined with other techniques that try to further reduce the state space, like symbolic model checking.

3

Towards Static POR for Probabilistic Systems

As mentioned before, in dynamic partial order reduction techniques, the checking of the fulfilment of the ample set conditions is usually performed while the state space is being generated. Nevertheless, it is possible to guarantee the satisfaction of some of those conditions carrying out a static analysis over the probabilistic control graphs, prior to state space generation. In this chapter, we elaborate on a static technique that allows us to guarantee conditions C2 and C3. The result of that hybrid, static and dynamic, technique is an ample set POR where conditions C2 and C3 over ample sets are checked statically using information from the individual probabilistic control graphs and where the rest of conditions are checked dynamically.

3.1. Sticky Set POR Technique

This technique, inspired in the static partial order for non-probabilistic systems, described in [KLM⁺98] and [BK08], is based in the identification of a set of *sticky actions* referred to as the *Sticky Set*. Those actions fulfil certain conditions:

- S1: Given Vis the set of all visible transitions, $Vis \subseteq Sticky$.
- S2: for any cycle s_0, s_1, \dots, s_n in \hat{M} , $\bigcup_{i \in [1, n]} ample(s_i) \cap Sticky \neq \emptyset$

Informally, condition S1 states that all visible transitions belong to the set of sticky transitions. S2 guarantees that no cycle on the reduced state space can be closed without executing a sticky transition. These conditions allow the enunciation of a new condition C3' on ample sets that replaces C2 and C3:

- C3': if $ample(s) \cap Sticky \neq \emptyset$ then $ample(s) = Act(s)$.

Finally, condition C3 implies that every state whose ample set contains a sticky transition is fully expanded. We can see how C3' suffices to guarantee previous condition C2, as every visible transition is also a sticky one. Regarding condition C3, it is guaranteed by the conjunction of S2 and C3', as in every end component $\xi = (T, Act_\xi)$ in the state space there exists at least one loop. This assessment is easy to see because, as ξ is an end component

is represented by a strongly connected subgraph, for every pair of states $(s_i, s_j) \in T \times T$ there is a path ρ from s_i to s_j . Analogously, there also exists a path ρ' from s_i to s_j as $(s_j, s_i) \in T \times T$. The path $\rho'' = \rho \cup \rho'$ represents a loop. Notice that even for the minimal end component where $|T|=1$, there exists at least a path $p = s_i \Rightarrow^\beta s_i$ as $(s_i, s_i) \in T \times T$. Therefore, as in every cycle there is at least an action $\alpha \in Sticky$, there is also at least one in every end component, hence having at least one fully expanded state.

3.2. Sticky Set Identification: the Naïve Approach

In order to identify a sufficient Sticky Set, we have implemented an algorithm that represents a naïve approach that does not generate the minimal, hence optimal, set of sticky transitions but one that fulfils conditions S1 and S2 previously stated. The pseudo-code notions of this algorithm, which we name *Naïve Sticky Set Identification Algorithm*, are presented in Figure.3.1. The algorithm starts defining a Sticky Set which only contains all the visible actions. After that, it performs a depth-first search on every probabilistic control graph of the system processes and keeps track of all loops contained in each of them (containing probabilistic actions or not). Every time a backward edge is found, the action it corresponds to is added to the Sticky Set.

As a matter of example, we consider again the *two-counters system* presented in Figure 2.3. We assume that the property to check only refers to variables $\{a, b\}$, then, the set of visible actions is $Vis = \{\beta, \epsilon\}$. If we apply the function *backwardActions* to each probabilistic control graph, we find that the backward edges are those corresponding to actions $\{\alpha, \delta, \beta, \epsilon\}$. Then, by executing the algorithm in Figure 3.1 we obtain the Sticky Set $Sticky = \{\alpha, \beta, \delta_1, \epsilon\}$. The new probabilistic control graphs with actions of the Sticky represented as dotted edges are presented in Figure 3.2. If we perform the interleaving in order to obtain the crossproduct probabilistic control graph, we obtain a new graph, depicted in 3.3, showing again the actions that belong to the Sticky Set as dotted edges. We can observe that it is impossible to close a loop in the new graph without going through a dotted edge, thus executing a Sticky Action. Then, upon generation of the state space corresponding to this crossproduct probability control graph, the check of conditions C2 and C3 is not necessary, as fully expanding systematically those states whose ample set contains a Sticky Transition guarantees them.

Theorem 3.2.1. Soundness of the Naïve Sticky Set Algorithm Given PCG_1, \dots, PCG_n and M the MDP they generate, if dynamic checks for conditions C2-C3 in state $s \in S'$ resume to checking $Sticky \cap ample(s) = \emptyset$, while the rest of conditions are checked dynamically, M' and M are stutter-trace-equivalent.

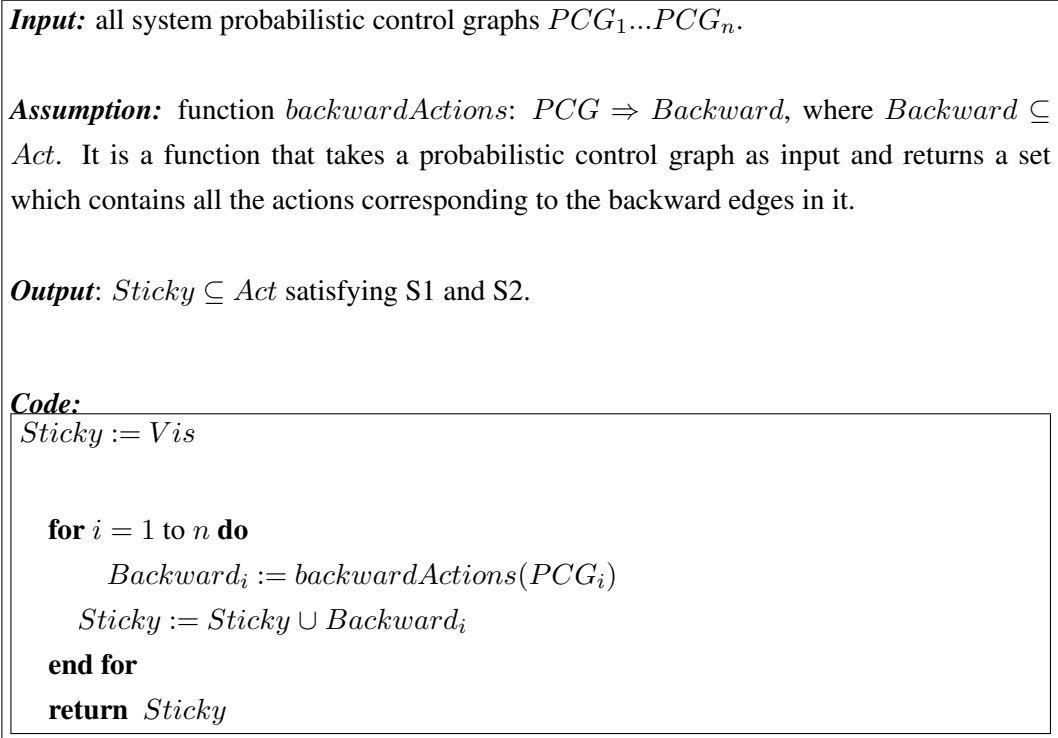


Figure 3.1: Naïve Sticky Set Identification Algorithm

Proof. In order to prove soundness of the Naïve Sticky Set Algorithm, we have to show that the Sticky Set $Sticky$ provided by it for MDP M fulfils conditions S1 and S2 stated before:

- **Fulfilment of S1:** it is obvious, as $Sticky$ is initially built up from the set of all visible actions. To such initial set some other actions are possibly added, but never removed, therefore always containing the whole visible actions set.
- **Fulfilment of S2:** every cycle in state space S derives from at least one control cycle $c = l \Rightarrow^{\alpha_1} s_1 \dots \Rightarrow^{\alpha_n} l_n$ such that $l = l_n$ and $n > 0$. Therefore, as the Naïve Sticky Set Algorithm adds to the Sticky Set every backward edge in the probabilistic control graph, every control cycle contains at least one Sticky Action. Then, we obtain that every cycle in the state space contains at least a Sticky Transition, whose origin state is fully expanded.

□

3.2.1. Theoretically Expected Improvement

As we mention before, the use of Naïve Sticky Set Algorithm prior to state space generation, thus in a static way, allows to guarantee that no check for conditions C2 and C3 has to

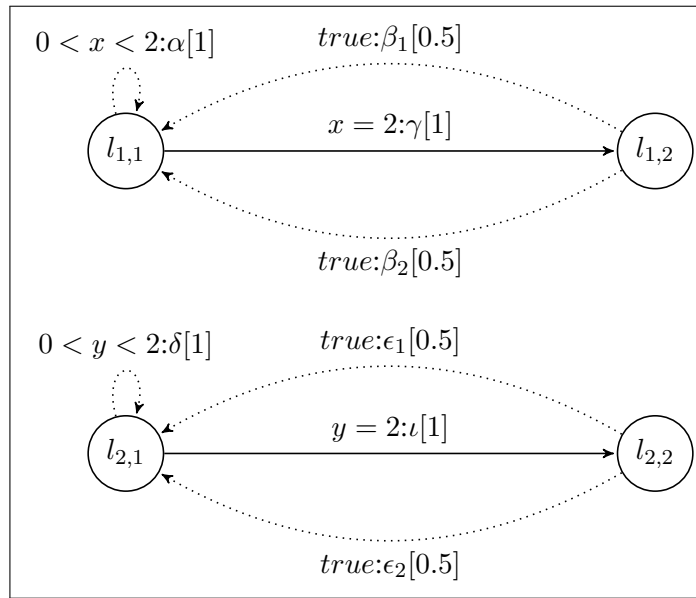


Figure 3.2: Sticky Actions in the individual Control Graphs of 2-counters system

be performed. In dynamic POR, condition C2 can be achieved just using local information in the vicinity, e.g. checking if the successors of an action change the value of the atomic propositions. Therefore, as all successors for every action are generated, the generation of unnecessary states, i.e. states $s \in S$ such that $s \notin \hat{S}$. Regarding fulfilment of C3, there exist different alternatives, like the ones implemented in [Cie11], which may require tracking all the paths for every cycle in order to perform some checks on it. The existence of a cycle in the probabilistic control graph of a single process can potentially generate a number of cycles in the state space which grows exponentially with the number of processes involved. Therefore, the computational complexity derived from the checks performed upon identification of every single cycle in the state space grows exponentially.

However, using a static algorithm, like Naïve Sticky Set Algorithm, checks for conditions C2 and C3 resume to a search in an list for every enabled action. Then, the computational complexity to identify an ample set that fulfils conditions C2-C3 is linearly proportional to the number of control locations in the whole system. Therefore, using Naïve Sticky Set Algorithm, we expect to achieve an improvement in the time of computing the ample sets, as less checks have to be performed in order to guarantee C2 and C3. Nevertheless, as this algorithm does not generate the smallest Sticky Set, a significant improvement in the state space reduction with respect to more sophisticated dynamic POR techniques is not expected. In the following section we provide a comparison over a set of experiments between the performance of an implementation of Naïve Sticky Set Algorithm in the model checker

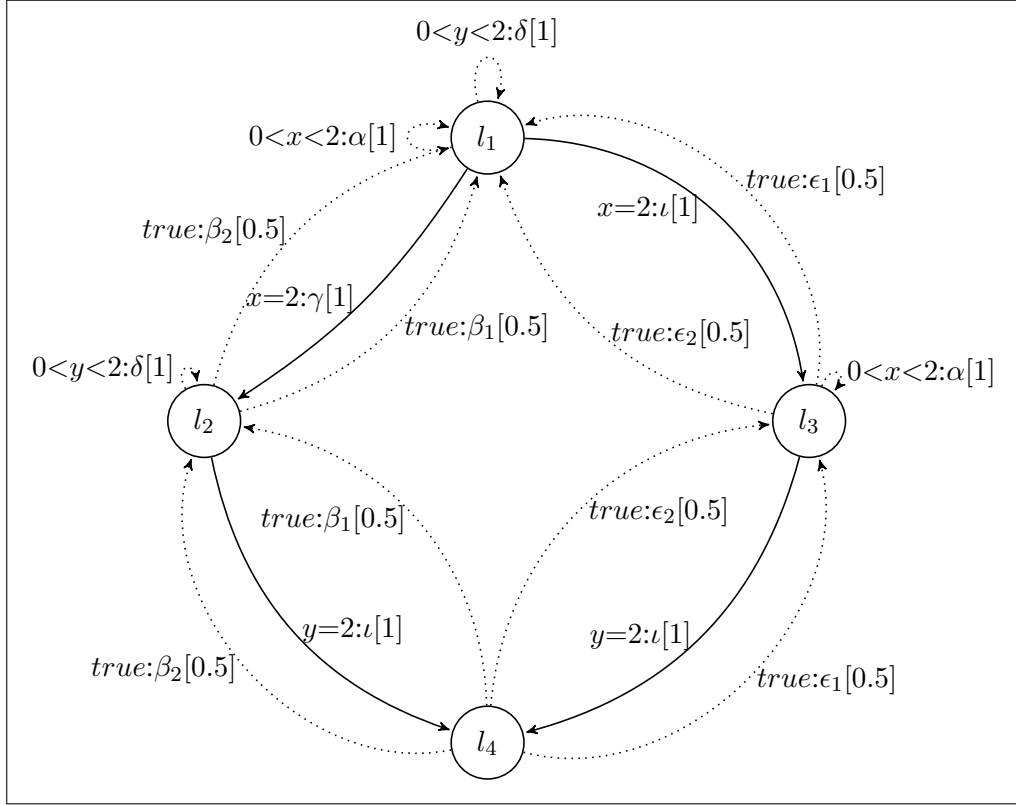


Figure 3.3: Crossproduct Probabilistic Control Graph for the 2-counters showing Sticky Set LiQuor [CC06] and some already implemented dynamic POR, described in [Cie11].

3.3. Experiments

After defining the Naïve Sticky Set Algorithm, we have implemented it in LiQuor Model Checker. In order to do that, we include a new stack data structure which keeps track of control states visited during initial Depth First Search (DFS) over each different probabilistic control graph. Every time a location l is revisited, the new stack structure is visited in search for the path composed by the states visited after the previous time l was visited in order to identify the possible loop existing over this location. If a loop is identified, the action that closes that loop, thus a backward edge, is marked added to a list of sticky actions. That list also contains all actions that modify a variable referred in the atomic propositions, i.e. visible actions. Then, during State Space Generation, when a new ample set is being generated, the membership of the different enabled actions to the list of sticky actions is checked in order to meet conditions C2 and C3. If any valid ample set identified must contain at least one of the actions in the list, the state is fully expanded.

After implementing the Naïve Sticky Set Algorithm in LiQuor, we have performed a set of empirical experiments with the aim of checking possible improvements acquired. In order to provide an intuition about such improvements, a comparison with different already implemented Dynamic POR techniques, described in [Cie11] as realizations R4-1 - R4-3, is presented. Those techniques are:

- **Fast:** automatically fully expands all states with an action that leads to a previous state.
- **Smart:** similar to previous one but does not expand fully a backward edge if the loop it closes already contains a fully expanded node.
- **Smarter:** only expands fully a backward edge if there exists at least an action α which is enabled in all the states contained in the loop it closes.

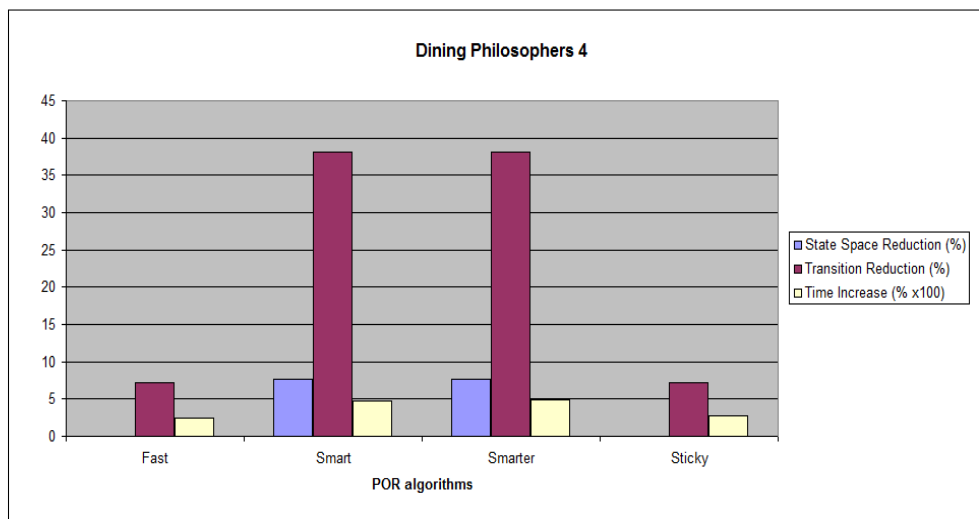


Figure 3.4: Statistics for four Dining Philosophers

Those experiments were performed for the well known models of the Dining Philosophers and Leader Election algorithms. In Figures 3.4 and 3.5, the percentage reduction in both the number of states and transitions is presented for a Dining Philosophers algorithm with four and five philosophers, respectively. We can see how Naïve Sticky Set Algorithm implementation does not suppose a reduction in the number of states generated, but it does suppose a reduction of 7, 3% in the number of transitions. Regarding the improvement with regards to the other algorithms, Naïve Sticky Set Algorithm behaves exactly as *Fast* algorithm but achieves a reduction lower than the one got by *Smart* and *Smarter* algorithms. Regarding the possible reduction in the time required to perform Model Checking over the model, we

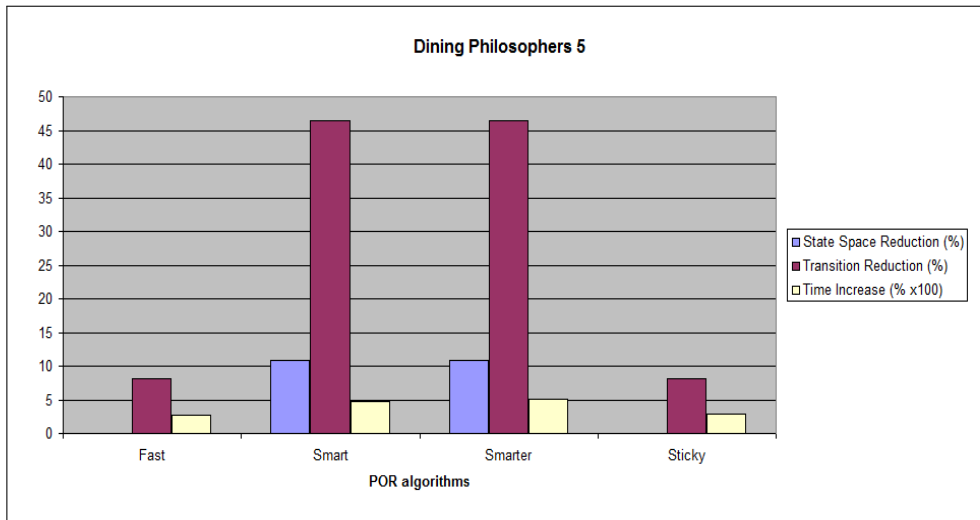


Figure 3.5: Statistics for five Dining Philosophers

can observe that in the case for four philosophers using Naïve Sticky Set Algorithm the time increases in a 286% with regards to the time required when POR is not performed. That increase is similar to the one for *Fast* algorithm and lower than ones for *Smart* and *Smarter* algorithms, which amount to an increase of 250%, 476% and 495% respectively. In the case for five Dining Philosophers, we can observe, again, how the reduction achieved by Naïve Sticky Set Algorithm is the same of that achieved by *Fast* one. The increase in the time required to perform Model Checking using each POR algorithm is also similar. With respect to *Smart* and *Smarter* algorithms, they suppose a higher reduction with a smaller time increase.

Figures 3.6 and 3.7 show information collected by repeating the experiment using a Leader Election algorithm with three and four processes respectively. In this case, we can see how Naïve Sticky Set Algorithm provides a significantly better reduction than the one provided with *Fast* algorithm. That reduction is even better, although slightly, than the one achieved by *Smart* algorithm. Regarding the time increase, it is higher than the one for both other algorithms, but very similar to the increase for *Smart* POR algorithm.

From these results we can state that it is possible to improve the reduction achieved by using Naïve Sticky Set Algorithm in some cases and, in any case, obtaining results as good as the ones achieved by using other naïve approaches that are more dynamic, like *Fast* algorithm. This irregular behaviour shows a dependency of the efficiency of the dynamic POR algorithms to the structure of the probabilistic control graphs.

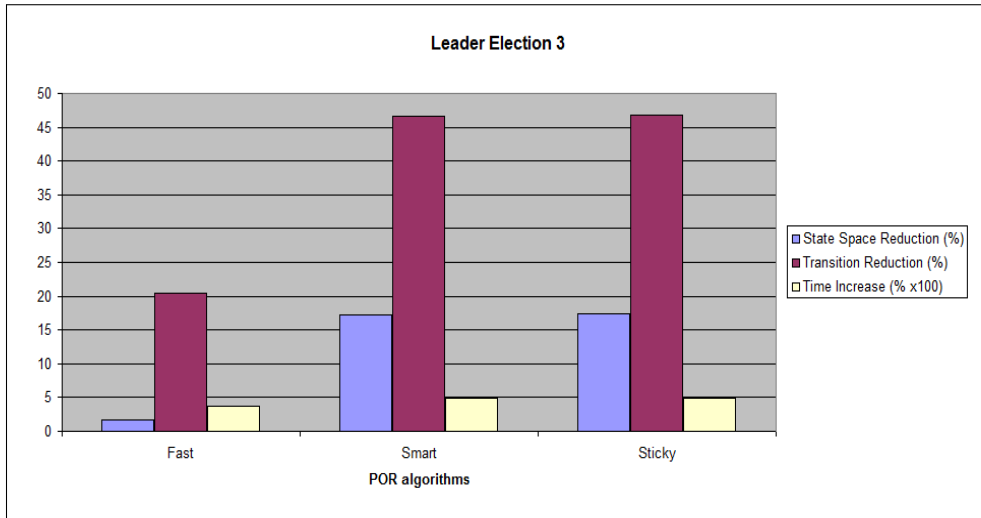


Figure 3.6: Statistics for a Leader Election between 3 processes

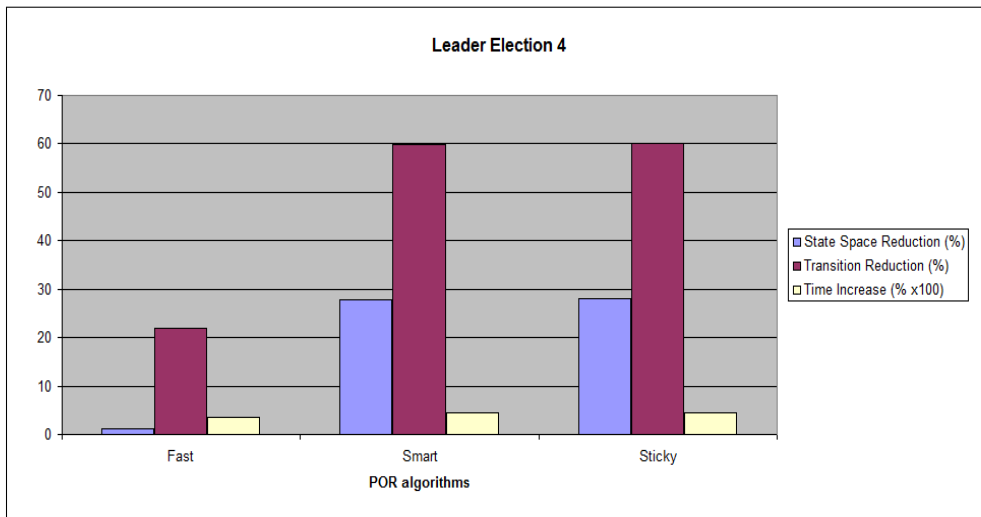


Figure 3.7: Statistics for a Leader Election among 4 processes

4

Completely Static POR of Probabilistic Systems

In the previous chapter we introduced an approach to be able to guarantee fulfilment of some of the ample set conditions statically, thus using only information from the individual probabilistic control graphs. By the development and implementation of that semi-static way of identifying valid ample sets, we have tried to increase the efficiency of the reduction process by means of a speedup derived from the reduction in the number of condition checks to be performed dynamically, i.e. during state space generation. In some cases, the reduction achieved was greater than the ones obtained with completely dynamic POR techniques. Therefore, it is possible to improve state space reduction by implementing static POR techniques. Moreover, such reduction can be further augmented by the combination of static POR and symbolic model checking, as suggested in [BK08] and [Pel98].

In this chapter we elaborate, first, on an already existing static POR technique for non-probabilistic systems, described in e.g. [BK08]. Then, we describe two new completely static POR techniques for both probabilistic and non-probabilistic systems, which take as basis the already mentioned one, only suitable for non-probabilistic systems. Finally we present some experiments performed after the implementation of both techniques into LiQuor Model Checker.

4.1. Static POR for Non-Probabilistic Systems

As we mention before, in [BK08] a technique for reducing the state space S of the transition system TS , which can be seen as an MDP where every transition has one and only one effect which is taken with probability 1, of a non-probabilistic system using only static data is described. Given n program graphs PG_i for $1 \leq i \leq n$, whose crossproduct PG would generate the transition system TS , the aforementioned technique modifies each single program graph PG_i into PG'_i , whose crossproduct is PG' . PG' generates a transition system TS' , whose state space is $S' \subseteq S$, equivalent to TS for all stutter invariant properties. The idea is that the reduction achieved relies on the static identification of valid ample sets.

In order to be able to start applying that technique, we have to calculate a valid sticky set

Sticky, i.e. one which fulfils conditions S1 and S2 mentioned in the previous chapter, and the dependency relation D , which contains all pairs of dependent actions. Once *Sticky* and D have been identified, we should also construct the digraph $G_i = \{V_i, E_i\}$ corresponding to each single Program Graph. In digraph G_i , each node $v \in V_i$ corresponds to a different location $l \in L_i$ and each edge $e \in E_i$ to a different action $\alpha \in Act_i$, hence not allowing the same action to appear more than once. We can obtain that by simply renaming actions. For the sake of simplicity, from now on, we refer to a transition from location l to location l' by execution of an action α with guard g as $l \rightarrow^{g:\alpha} l'$. Then, the procedure to follow is:

1. Every edge $l \rightarrow^{g:\alpha} l'$ such that $\alpha \in Sticky$ is marked as *sticky*.
2. An edge $l_i \rightarrow^{g_i:\alpha} l'_i$ in PG_i is marked as *good* if and only if there does not exist an edge $l_j \rightarrow^{g_j:\beta} l'_j$ in PG_j where $i \neq j$ such that $(\alpha, \beta) \in D$ and the variables modified by β are not referred in g .
3. Location l in PG_i is marked as *ample* if and only if all its outgoing edges $l \rightarrow^{g:\alpha} l'$ are marked as *good*, and not *sticky*, and the disjunction of its guards is a tautology.
4. Create n new boolean global variables $ample_1, \dots, ample_n$. These variables will be updated atomically in every transition.
5. Every edge $l_i \rightarrow^{g_i:\alpha} l'_i$ is replaced with $l_i \rightarrow^{g_i:\alpha'} l'_i$, where α' is the atomic execution of α followed by $ample_i = true$ if l_i is marked as *ample* or followed by $ample_i = false$ otherwise.
6. Calculate new propositions, referring to variables $ample_i$, which will strengthen the guards of every edge:
 - $h_i = \bigwedge_{1 \leq j < i} \neg ample_j \wedge ample_i$. These propositions will be added to the guards of actions that correspond to a valid ample set. The goal is to build an ample set containing only the actions of a single location $l \in PG_i$ marked as *ample*, giving preference to those belonging to a Program Graph PG_i with lower index i .
 - $f = \bigwedge_{1 \leq j \leq n} \neg ample_j$. This proposition will appear in locations that correspond to a fully expanded state, thus allowing every action to be enabled.
7. Every edge $l_i \rightarrow^{g_i:\alpha} l'_i$ is replaced with $l_i \rightarrow^{g'_i:\alpha} l'_i$ where $g'_i = h_i \wedge g$ if l_i is marked as *ample* and $g'_i = f \wedge g$ otherwise.

8. Initial conditions $g_{0,i}$ of every Program Graph are extended into $g'_{0,i} = g_{0,i} \wedge \text{ample}_i$ if initial location $l_{0,i}$ of PG_i is marked as *ample* or into $g'_{0,i} = g_{0,i} \wedge \neg \text{ample}_i$ if it is not.

All these steps turn original program graphs PG_1, \dots, PG_n into PG'_1, \dots, PG'_n which will result in reduced TS' . We remark that there is no reduction in the individual program graphs, as every location and transition is conserved, though the last ones are modified.

In order to clarify this technique, we provide an example of its application, extracted from [BK08]. Consider two processes that implement the behaviour of an iterative 1 to 10 counter (although upper bound of the counter is not relevant for the example and could be abstracted to N) which modify a boolean variable after every iteration. The Program Graphs PG_1 and PG_2 corresponding to each counter are shown in Figure 4.1.

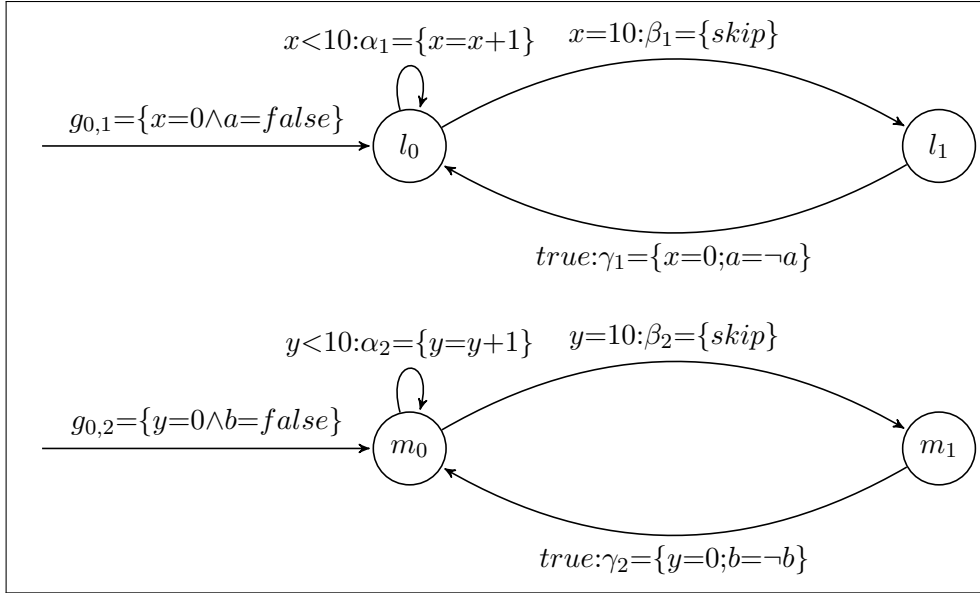


Figure 4.1: Program Graphs of the iterative counters

Assuming the set AP of atomic propositions only refer to boolean variables a and b , the set of visible actions is $Vis = \{\gamma_1, \gamma_2\}$. If we choose the Sticky Set to be $Sticky = Vis$, it fulfils conditions S1 and S2, as it contains all visible actions and every loop in the state space will contain at least one sticky transition. As every action and guard only refers to local variables, the set of dependent actions is $D = \emptyset$. Then, we mark as *sticky* the edges corresponding to actions γ_1 and γ_2 and as *good* all other edges, as actions α_1 , β_1 and γ_1 are independent to actions α_2 , β_2 and γ_2 . After labelling the edges, we label locations m_0 and l_0 as *ample* as all of their outgoing transitions are marked as *good* and the disjunction of their guards is a tautology (we must notice the domains of variables x and y are

$Dom(x) = Dom(y) = [0, 10]$). Then, we calculate $f = \neg ample_1 \wedge \neg ample_2$ and, as there are two processes, we compute $h_1 = ample_1$ and $h_2 = \neg ample_1 \wedge ample_2$. Once the new propositions are calculated, we extend guards of γ_1 and γ_2 with f , as they come from a location not marked as *ample*, guards of α_1 and β_1 with h_1 , as they both originate from a location marked as *ample*, and guards of α_2 and β_2 with h_2 for the same reason. After guard strengthening, we atomically extend actions α_1 and γ_1 with expression $ample_1=true$, actions α_2 and γ_2 with $ample_2=true$, action β_1 with expression $ample_1=false$ and action β_2 with $ample_2=false$. Finally, as initial locations l_0 and m_0 are both marked as *ample*, initial conditions $g_{0,1}$ and $g_{0,2}$ are extended with a conjunction to proposition $ample_1=true$ and $ample_2=true$ respectively. The result of the whole process are the modified program graphs displayed in Figure 4.2.

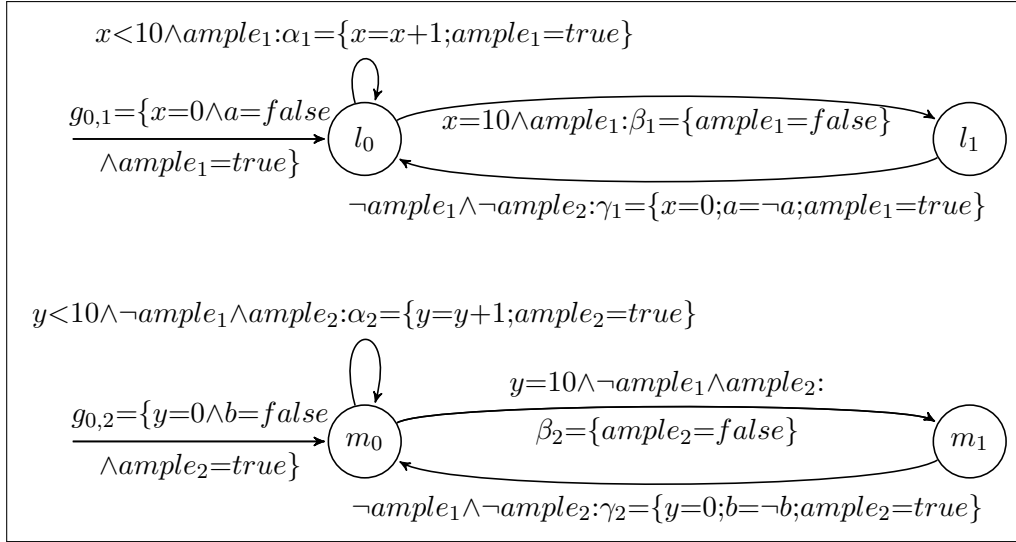


Figure 4.2: Modified Program Graphs of the iterative counters

As it is proven in [BK08], the new transition system TS' derived from the modified program graphs PG'_i is such that TS' and TS are stutter-trace-equivalent, as the reduction is performed by identifying valid ample sets for non-probabilistic systems. This statement also implies that, by following described steps, conditions C0, C1 and C3' are guaranteed, and as the system is not probabilistic, fulfilment of conditions C4 or C4' is not needed.

4.2. Static POR for Probabilistic Systems

As we state before, the ample sets identified by the Static POR technique described in the previous chapter guarantee the identification of ample sets that fulfil conditions C0-C3', where C3' suffices for C2-C3. Nevertheless, neither condition C4, which imposes the re-

quirement to find either a singleton ample set or have a fully expanded state nor condition C4' which imposes the singleton ample set whenever a probabilistic action can be executed before the actions in the ample set, are guaranteed by the aforementioned technique. Therefore, that technique cannot be applied to perform a reduction of the state space of a probabilistic system. In order to be able to perform a Completely-Static Partial Order Reduction of the state space of a probabilistic system, we have extended that strategy in order to be able to statically guarantee conditions C4 and C4'. Before we start the description of such extension, there are some considerations that should be taken into account, as well as some preliminary computations.

4.2.1. Fundamental Considerations

The following considerations are assumed in order to extend the original non-probabilistic Static POR technique to the probabilistic setting:

Consideration 1: Omitting an ample set is safe. By this assessment, we refer to the fact that when the correctness of an ample set candidate is not guaranteed, but its incorrectness is not either, the choice to consider such candidate to be a not valid ample set is sound. It is obvious, as discarding valid reduced ample sets only supposes a decrease in the reduction achieved but, as no transition is omitted, guarantees similarity to the original unreduced system.

Consideration 2: Notations for Probabilistic Control Graphs. In order to refer the different sets of elements in a probabilistic control graph $PCG=(Loc, Event, \rightsquigarrow, l_0)$ over a set of variables Var and communication channels $Chan$, from now on, we use the following notations:

- The set of possible events, i.e. those with a probability greater than zero, corresponding to the execution of an action $\alpha \in Act$ is $Events(\alpha)$.
- $written(\alpha) = \{a \in Var \cup Chan \mid \exists \eta \in Dom(a), e \in Events(\alpha). \eta \neq Effect(e, \eta)\}$ is the set of variables modified by the effects of action α . Analogously, the set of variables read by those effects is $read(\alpha)$.
- Similarly, $written(Act) = \bigcup_{\alpha \in Act} written(\alpha)$ is the set of all variables modified by the set of actions Act of PCG . Again, the set of variables read by those actions is $read(Act)$.

- Regarding the guard g of action α , the set of variables referenced by the propositions of the guard is $varsGuard(\alpha)$.
- In order to refer to all actions modified and read by an action α and its guard g , we write $accessed(\alpha) = \{written(\alpha) \cup read(\alpha) \cup varsGuard(\alpha)\}$. Similarly, the set of all variables accessed by PCG is $accessed(Act) = \{written(Act) \cup read(Act) \cup varsGuard(Act)\}$

Consideration 3: On the structure of Probabilistic Actions. This kind of actions are notable for having different possible effects. From the effects corresponding to a single probabilistic action, only one is triggered every time that action is executed. The triggered effect is chosen stochastically with respect to some fixed probabilistic distribution. Therefore, it suffices for an action to have one of its effects making it dependent to another action, i.e. given effects $|Events(\alpha)| \geq 1$ and there exists $e \in Events(\alpha)$ such that execution of e can affect the enabling of another action β , then α and β are dependent disregarding the possibility that every other effect $f \in Event(\alpha) \setminus e$ would not make them dependent. Nevertheless, as the Static POR refers to atomic extension of edges, such extension is only performed in the effect that the edge depicts, leaving the rest of effects from the same action untouched. Besides, as all of those effects belong to the same action, all of them share the same guard statement. Then, if we strengthen the guard for one of the effects, it is also extended for all the rest.

4.2.2. Preliminary Calculations

As happened in the case of the non-probabilistic version, in order to start applying the static reduction we have to calculate the set *Sticky* and the dependency relation D . The former, can be calculated, for instance, using *Naïve Sticky Set Algorithm* presented in Chapter 2. Regarding D , we calculate it by the following conservative overapproximation, extracted from [BK08]:

Dependency Relation Overapproximation. Given $Act = \bigcup_{1 \leq i \leq n} Act_i$ the complete set of actions of a system, either probabilistic or non-probabilistic, composed by n processes with Program Graphs PG_1, \dots, PG_n , the dependency relation $D \subseteq Act \times Act$ is such that for actions $\alpha \in Act_i$ and $\beta \in Act_j$ where $i \neq j$, $(\alpha, \beta) \in D$ if and only if at least one of the following statements hold:

- $written(\alpha) \cap accessed(Act_j) \neq \emptyset$.

- $written(Act_j) \cap varsGuard(\alpha) \neq \emptyset$.
- $\exists c \in Chan, \gamma \in Act_j, \delta \in Act_i$ such that $c \in accessed(\gamma) \cap accessed(\delta)$

Which, less formally, are equivalent to:

- Some effect of α changes valuation of a variable referenced, i.e. written or read, by an action $\gamma \in Act_j$.
- There exists some action $\gamma \in Act_j$ that modifies some of the variables in the guard g of α .
- There exists a communication channel c that is accessed, for reading or writing, by actions $\delta \in Act_i$ and $\gamma \in Act_j$.

As we can see, this overapproximation allows the situation where actions $\alpha \in Act_i$ and $\beta \in Act_j$ where $i \neq j$ that do not even share a common variable, $accessed(\alpha) \cap accessed(\beta) = \emptyset$, are considered as dependent, $(\alpha, \beta) \in D$, because there is some other action $\gamma \in Act_j$ which does, e.g. $written(\alpha) \cap accessed(\gamma) \neq \emptyset$, then $(\alpha, \gamma) \in D$. The main rationale behind this approach is that the overapproximation takes into account the possibility that the execution of α enables action γ , which is dependent to β as $\beta, \alpha \in Act_j$. It is not possible to discard that possibility just by using local criteria, then α and β are considered dependent.

4.2.3. Naïve SPOR

As the need of extension derives from the necessity to statically guarantee condition C4, recalling:

C4: if $ample(s) \neq Act(s)$ then $|ample(s)| = 1$.

Then, we could achieve this goal by strengthening the condition to consider a location as *ample*. We impose it to have one and only one outgoing transition. Therefore, step 3 from page 30 must be replaced with the following step 3':

3'. Location l in PG_i is marked as *ample* if and only if it has one and only one outgoing transition, this transition is marked as *good* and its guard is a tautology.

Lemma 4.2.1. Soundness of Naïve SPOR

Given PCG_1, \dots, PCG_n which generate MDP M and applying Naïve SPOR, we get PCG'_1, \dots, PCG'_n whose related MDP M' is such that for every stutter-invariant time property E expressed in $PCTL^*_{-\circ}$, $M \models E \Leftrightarrow M' \models E$.

Proof. In order to prove it, we must show that the new ample sets fulfil conditions C0-C4. We prove this by reasoning on the extension performed. As every location marked as *ample* will have a single outgoing action, and the algorithm imposes that ample sets will only contain actions from a single probabilistic control graph, every ample set will have cardinality one. Therefore, either $\text{ample}(s) = \text{Act}(s)$ or $|\text{ample}(s)| = 1$, which corresponds to the *Branching Condition* introduced in [GKPP99]. Then, fulfilment of condition C4 is obvious. Regarding the rest of conditions, we must notice that the extension can keep or decrease, never increase, the number of ample sets with respect to the non-probabilistic Static POR, being the former always contained in the latter. The reason for this derives from the fact that the location labelling is maintained from the technique described in page 30. Therefore, taking into account soundness of non-probabilistic Static POR provided in [BK08] and Consideration1 in page 33, fulfilment of conditions C0-C3 directly follows. \square

Even though Naïve SPOR is correct, it strongly limits the potential reduction achieved, as will only allow singleton ample sets. This approach is sufficient for branching-time stutter invariant properties, as shown in [GKPP99]. Nevertheless, in the setting of linear-time stutter invariant properties, this shortcoming becomes more noticeable if we apply the new technique in a non-probabilistic system and compare it with the results obtained using non-probabilistic Static POR. The comparison of both approaches is presented in Figure 4.4. In this figure, we present a system composed by two programs. All actions of each program are independent from the actions of the other, then having $D = \emptyset$, and their guard is always a tautology. There are only two Sticky actions, corresponding to the two backward edges. If we apply Naïve SPOR in such a system, we can only identify three ample locations, l_1, l_2 and m_2 shown in dotted lines, whose only action represents a valid ample set from states that include them. Compared with the results obtained if we use non-probabilistic Static POR technique, which is possible as it is a non-probabilistic system, we identify seven ample locations instead. As we can see, Naïve SPOR is not efficient enough to be applied in the non-probabilistic setting.

Another shortcoming of Naïve SPOR, which also derives from its lack of ability to check if there exists any reachable probabilistic action in the system, can be observed in probabilistic systems like the one shown Figure 4.4. This system is composed by two programs. Again, actions from one probabilistic control graph are independent of those from the other and their guards are a tautology, hence being always enabled. There is only one probabilistic action δ with two effects δ_1 and δ_2 which have the same 0.5 probability. There are also three sticky actions that correspond to backward edges δ , κ and σ . As we can see, if we apply Naïve SPOR technique on the system, we identify four ample locations, l_1, l_5, m_1 and m_2 .

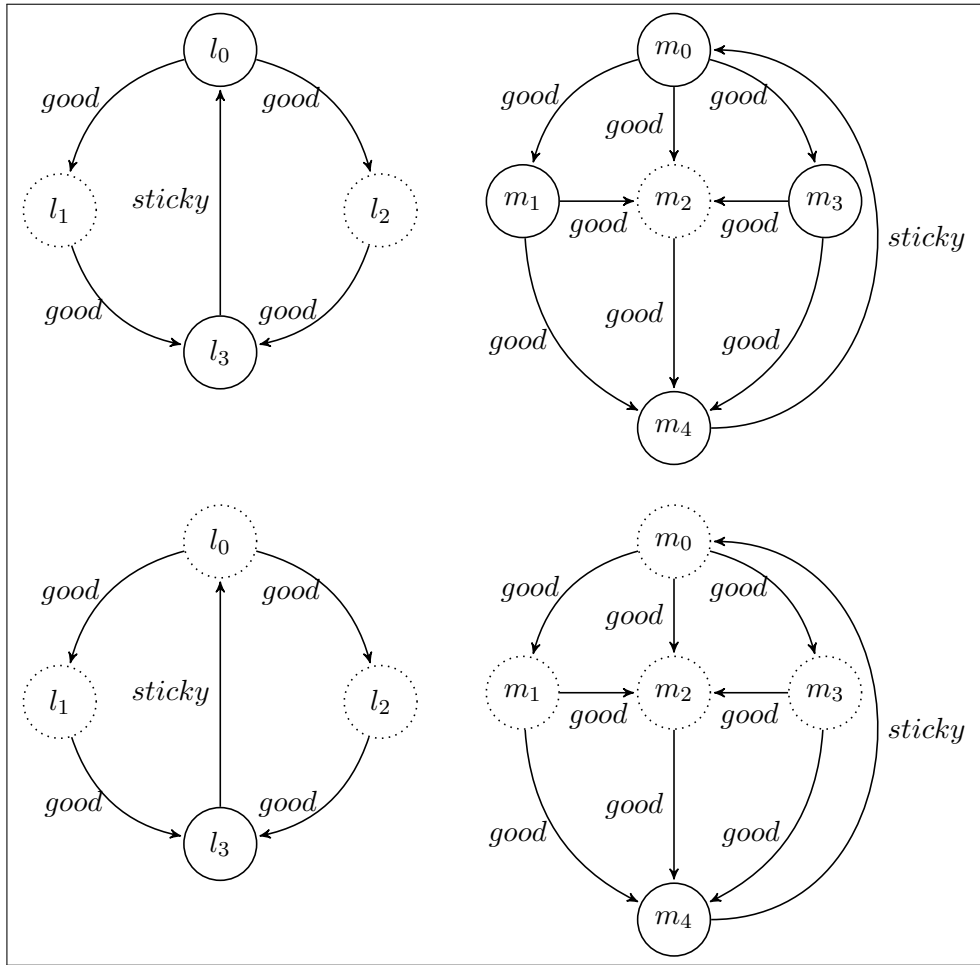


Figure 4.3: Comparison between Naïve SPOR (above) and non-probabilistic SPOR (below)

Nevertheless, upon execution of action β , there are no more reachable probabilistic actions in the system, which implies that, from that point, condition C4' always hold. Then, a more efficient SPOR technique, like the one we have developed and present in next section, would also label locations l_3 , l_4 and m_0 as *ample*, hence increasing the reduction achieved. However, we must take into account that as the stronger condition C4 is not guaranteed, the resulting reduced system could not be checked for branching time properties, but for linear time ones, both qualitative and quantitative.

4.2.4. Reachability-Aware SPOR

As we show in the previous section, Naïve SPOR technique, in favor of always ensuring fulfilment of ample set condition C4, disregards the situation in which there are no more reachable probabilistic actions. In such situation, the singleton ample set restriction is no

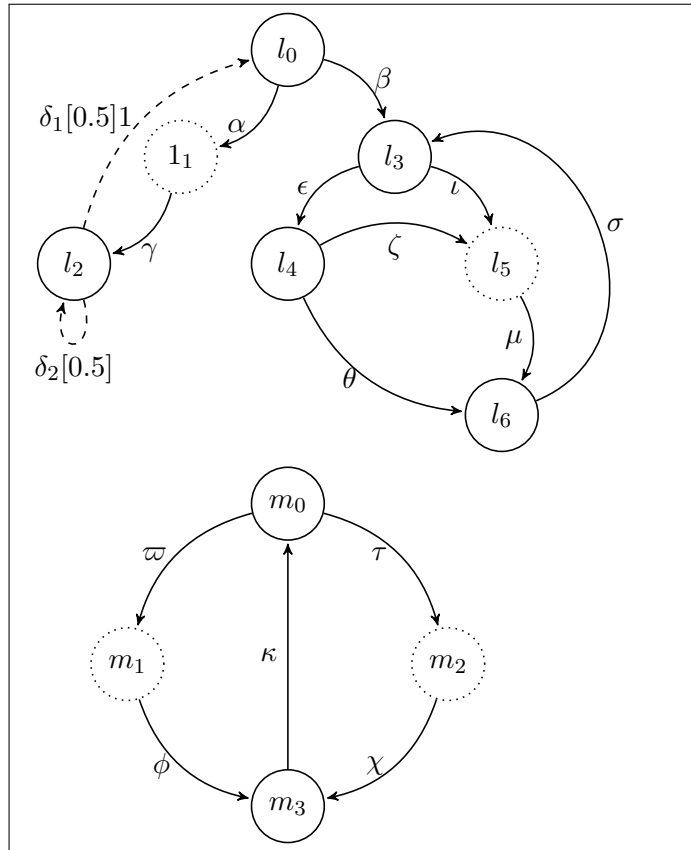


Figure 4.4: Yet another shortcoming of Naïve SPOR

longer necessary to comply with restriction C4'. In order to also consider that kind of situations, we have extended Naïve SPOR technique further. The idea is to identify those locations which would have been marked as *ample* by the non-probabilistic SPOR but are not because they have more than one outgoing action, even though it is not possible to execute any probabilistic action in the system. We must remark that such lack of possibility does not only refer to reachability of a probabilistic action within the same Probabilistic Control Graph, but from all others. Therefore, those locations, from now on referred as *conditionally ample locations*, are not always considered as ample locations, but when it is not possible to execute any probabilistic action by any of the programs in the system. In order to cope with this reachability problem, and considering these ideas, we have developed a new technique for *Reachability-Aware SPOR*, which consists of the following steps:

1. Every edge $l \rightarrow^{g:\alpha} l'$ such that $\alpha \in Sticky$ is marked as *sticky*.
2. An edge $l_i \rightarrow^{g_i:\alpha} l'_i$ in PG_i is marked as *good* if and only if there does not exist an edge $l_j \rightarrow^{g_j:\beta} l'_j$ in PG_j where $i \neq j$ such that $(\alpha, \beta) \in D$.

3. A location l in PG_i is marked as *probabilistic* if there exists a path $\rho = l \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} l^{n-1} \xrightarrow{\beta} l^n \dots$ is a path in PG_i where β is a probabilistic action. The motivation of this labelling is finding all locations from where it is possible to reach a probabilistic action.
4. Every location $l \in PG_i$ is marked with one or none of these labels:
 - **ample** if and only if it has only one outgoing edge $l \xrightarrow{g:\alpha} l'$ marked as *good*, and not *sticky*, and its guard g is a tautology.
 - **cond_ample** if and only if all its outgoing transitions, whose quantity has to be greater than one, are marked as *good* and the disjunction of their guards is a tautology. These locations are the aforementioned *conditionally ample locations*.
5. Create n new boolean global variables $ample_1, \dots, ample_n$. These variables will be updated atomically in every transition.
6. Create n new boolean global variables $prob_1, \dots, prob_n$. Each of these variables can be updated once or never in the whole system.
7. Create n new boolean global variables $cond_ample_1, \dots, cond_ample_n$. These variables will be updated atomically in every transition.
8. Every edge $l_i \xrightarrow{g_i:\alpha} l'_i$ in PCG_i is replaced with $l_i \xrightarrow{g_i:\alpha'} l'_i$, where α' is the atomic execution of α followed by $ample_i = true$ if l_i is marked as *ample* or followed by $ample_i = false$ otherwise.
9. Every edge $l_i \xrightarrow{g_i:\alpha'} l'_i$ in PCG_i is replaced with $l_i \xrightarrow{g_i:\alpha''} l'_i$, where α'' is the atomic execution of α' followed by $cond_ample_i = true$ if l_i is marked as *cond_ample* or followed by $cond_ample_i = false$ otherwise.
10. Every edge $l_i \xrightarrow{g_i:\alpha''} l'_i$ in PCG_i where l_i is marked as *probabilistic* and l'_i is not, is replaced with $l_i \xrightarrow{g_i:\alpha'''} l'_i$, where α''' is the atomic execution of α'' followed by $prob_i = false$.
11. Calculate new propositions, referring to variables $ample_i$, $cond_ample_i$ and $prob_i$, which will strengthen the guards of every edge:
 - $h_i = \bigwedge_{1 \leq j < i} \neg ample_j \wedge ample_i$. As happened in the original technique for non-probabilistic systems, the goal of this proposition is to build an ample set containing only the actions of a single location $l \in PG_i$ marked as *ample*, giving preference to those belonging to a Program Graph PG_i with lower index i .

- $f = \bigwedge_{1 \leq j \leq n} \neg ample_j$. When this proposition holds, there is no location marked as *ample*, thus trying to build an ample set with either the actions of some locations marked as *cond_ample* or fully expanding the state.
 - $k_i = \bigwedge_{1 \leq j < i} \neg cond_ample_j \wedge cond_ample_i$. Similarly to variable h_i , this variable will aim at building ample sets containing the actions of a single location, this time a *conditionally ample location*.
 - $r = \bigwedge_{1 \leq j \leq n} \neg cond_ample_j$. Analogously to proposition f , this proposition holds when the state does not contain any location marked as *cond_ample*.
 - $p = \bigvee_{1 \leq j \leq n} (prob_j \wedge \neg cond_ample_j)$. This proposition holds when at least one of the control locations of the state is marked as *probabilistic* but not as *cond_ample*.
 - $q = \bigwedge_{1 \leq j \leq n} \neg prob_j$. This proposition holds when none of the control locations of the state is marked as *probabilistic*.
12. Every edge $l_i \xrightarrow{g_i:\alpha} l'_i$ in PCG_i is replaced with $l_i \xrightarrow{g'_i:\alpha} l'_i$ where g'_i , depending on the labelling of l_i , will be:
- $g'_i = h_i \wedge g_i$ if l_i is marked as *ample*.
 - $g'_i = f \wedge ((cond_ample_i \wedge prob_i) \vee (k_i \wedge q) \vee p) \wedge g_i$ if l_i is marked as *cond_ample*.
 - $g'_i = f \wedge (p \vee r) \wedge g_i$ if l_i is not marked as *ample* or *cond_ample*.
13. Initial conditions $g_{0,i}$ of every Probabilistic Control Graph are extended with three new propositions, $g'_{0,i} = g_{0,i} \wedge a_i \wedge b_i \wedge c_i$, which will be:
- $a_i = ample_i$ if initial location $l_{0,i}$ is marked as *ample* and $a_i = \neg ample_i$ otherwise.
 - $b_i = cond_ample_i$ if $l_{0,i}$ is marked as *cond_ample* and $b_i = \neg cond_ample_i$ otherwise.
 - $c_i = prob_i$ if $l_{0,i}$ is marked as *probabilistic* and $c_i = \neg prob_i$ otherwise.

In order to clarify how this new technique works, we provide an example. Consider the probabilistic system depicted in Figure 4.4 in page 38, where, as it is already mentioned, dependency relation $D = \emptyset$ and $Sticky = \{\delta, \kappa, \sigma\}$. As first step, we have to mark the four edges of actions δ, κ and σ as *sticky* (notice there are two edges corresponding to probabilistic sticky action δ) and all other actions as *good*. Then, we have to identify all locations

from where at least one probabilistic action is reachable. A simple way to do this is starting by marking as *probabilistic* all locations which have at least one probabilistic outgoing transition. From each of those locations, we have to propagate backwards, with respect to transition direction, *probabilistic* label, hence marking as *probabilistic* every location l_i such that there is a transition $l_i \xrightarrow{g_i:\alpha} l'_i$ where l'_i is already marked as *probabilistic*. This backward propagation process is iterated until no more new locations are labelled. In our example, we start by labelling location l_2 as *probabilistic*. Then, we propagate that mark to location l_1 , as there is an edge $l_1 \xrightarrow{g:\gamma} l_2$ and, finally, from l_1 to l_0 from edge corresponding to action α . Then, as final step of the labelling process, we mark locations l_1, l_5, m_1 and m_2 as *ample* and locations l_0, l_3, l_4 and m_0 as *cond_ample*.

After the labelling process is complete, we start extending the guards and effects of the different actions. As their corresponding edges lead to a location marked as *ample*, actions $\alpha, \iota, \zeta, \tau$ and ϖ have their effects atomically extended with the assignment $ample_i = true$, where $i \in [1, 2]$ is the index of the probabilistic control graph the action belongs to. All other edges are extended with variable assignment $ample_i = false$, taking into account that this assignment occurs twice for probabilistic action δ as it has two possible effects. In a similar way, actions $\beta, \epsilon, \sigma, \kappa$ and effect δ_1 of action δ are extended with assignment $cond_ample_i = true$ while all the rest of edges are extended with assignment $cond_ample_i = false$. As location l_0 is marked as *probabilistic* but not its successor location l_3 , we atomically extend the effect of action β that transits from the former to the latter with variable assignment $prob_1 = false$. Then, we calculate propositions:

- $h_1 = ample_1$
- $h_2 = \neg ample_1 \wedge ample_2$
- $f = \neg ample_1 \wedge \neg ample_2$
- $k_1 = cond_ample_1$
- $k_2 = \neg cond_ample_1 \wedge cond_ample_2$
- $r = \neg cond_ample_1 \wedge \neg cond_ample_2$
- $p = (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
- $q = \neg prob_1 \wedge \neg prob_2$

Once these propositions are computed, we strengthen the guards of actions γ, μ, ϕ and χ by conjunctively adding proposition h_i to them. To the guards of actions $\alpha, \beta, \epsilon, \iota, \zeta, \theta, \varpi$ and

τ we add condition $f \wedge ((prob_i \wedge cond_ample_i) \vee (k_i \wedge q) \vee p)$ and to the guards of actions who have not been strengthened yet the condition $f \wedge (p \vee r)$. Finally, initial condition $g_{0,1}$ is extended with condition $\neg ample_1, \wedge \neg cond_ample_1 \wedge prob_1$ and initial condition $g_{0,2}$ with proposition $\neg ample_2, \wedge cond_ample_2 \wedge \neg prob_2$. Therefore, resulting set of actions and initial conditions is:

- $g'_{0,1} = g_{0,1} \wedge \neg ample_1, \wedge cond_ample_1 \wedge prob_1$
- $g'_{0,2} = g_{0,2} \wedge \neg ample_2, \wedge cond_ample_2 \wedge \neg prob_2$
- $\alpha' = g'_\alpha: [1]\alpha'_1$
 - $g'_\alpha = g_\alpha \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((prob_1 \wedge cond_ample_1) \vee (cond_ample_1 \wedge (\neg prob_1 \wedge \neg prob_2))) \vee (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
 - $\alpha'_1 = \alpha_1; ample_1 = true; cond_ample_1 = false$
- $\beta' = g'_\beta: [1]\beta'_1$
 - $g'_\beta = g_\beta \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((prob_1 \wedge cond_ample_1) \vee (cond_ample_1 \wedge (\neg prob_1 \wedge \neg prob_2))) \vee (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
 - $\beta'_1 = \beta_1; ample_1 = false; cond_ample_1 = true; prob_1 = false$
- $\gamma' = g'_\gamma: [1]\gamma'_1$
 - $g'_\gamma = g_\gamma \wedge ample_1$
 - $\gamma'_1 = \gamma_1; ample_1 = false; cond_ample_1 = false$
- $\delta' = g'_\delta: \{[0.5]\delta'_1, [0.5]\delta'_2\}$
 - $g'_\delta = g_\delta \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((\neg cond_ample_1 \wedge \neg cond_ample_2) \vee (prob_1 \wedge cond_ample_1) \vee (prob_2 \wedge cond_ample_2))$
 - $\delta'_1 = \delta_1; ample_1 = false; cond_ample_1 = true$
 - $\delta'_2 = \delta_2; ample_1 = false; cond_ample_1 = false$
- $\epsilon' = g'_\epsilon: [1]\epsilon'_1$
 - $g'_\epsilon = g_\epsilon \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((prob_1 \wedge cond_ample_1) \vee (cond_ample_1 \wedge (\neg prob_1 \wedge \neg prob_2))) \vee (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
 - $\epsilon'_1 = \epsilon_1; ample_1 = false; cond_ample_1 = true$

- $\iota' = g'_\iota : [1] \iota'_1$
 - $g'_\iota = g_\iota \wedge \neg \text{ample}_1 \wedge \neg \text{ample}_2 \wedge ((\text{prob}_1 \wedge \text{cond_ample}_1) \vee (\text{cond_ample}_1 \wedge (\neg \text{prob}_1 \wedge \neg \text{prob}_2)) \vee (\text{prob}_1 \wedge \neg \text{cond_ample}_1) \vee (\text{prob}_2 \wedge \neg \text{cond_ample}_2))$
 - $\iota'_1 = \iota_1; \text{ample}_1 = \text{true}; \text{cond_ample}_1 = \text{false}$
- $\kappa' = g'_\kappa : [1] \kappa'_1$
 - $g'_\kappa = g_\kappa \wedge \neg \text{ample}_1 \wedge \neg \text{ample}_2 \wedge ((\neg \text{cond_ample}_1 \wedge \neg \text{cond_ample}_2) \vee (\text{prob}_1 \wedge \text{cond_ample}_1) \vee (\text{prob}_2 \wedge \text{cond_ample}_2))$
 - $\kappa'_1 = \kappa_1; \text{ample}_2 = \text{false}; \text{cond_ample}_2 = \text{true}$
- $\mu' = g'_\mu : [1] \mu'_1$
 - $g'_\mu = g_\mu \wedge \text{ample}_1$
 - $\mu'_1 = \mu_1; \text{ample}_1 = \text{false}; \text{cond_ample}_1 = \text{false}$
- $\phi' = g'_\phi : [1] \phi'_1$
 - $g'_\phi = g_\phi \wedge \text{ample}_2$
 - $\phi'_1 = \phi_1; \text{ample}_2 = \text{false}; \text{cond_ample}_2 = \text{false}$
- $\varpi' = g'_\varpi : [1] \varpi'_1$
 - $g'_\varpi = g_\varpi \wedge \neg \text{ample}_1 \wedge \neg \text{ample}_2 \wedge ((\text{prob}_2 \wedge \text{cond_ample}_2) \vee ((\neg \text{cond_ample}_1 \wedge \text{cond_ample}_2) \wedge (\neg \text{prob}_1 \wedge \neg \text{prob}_2)) \vee (\text{prob}_1 \wedge \neg \text{cond_ample}_1) \vee (\text{prob}_2 \wedge \neg \text{cond_ample}_2))$
 - $\varpi'_1 = \varpi_1; \text{ample}_2 = \text{true}; \text{cond_ample}_2 = \text{false}$
- $\sigma' = g'_\sigma : [1] \sigma'_1$
 - $g'_\sigma = g_\sigma \wedge \neg \text{ample}_1 \wedge \neg \text{ample}_2 \wedge ((\neg \text{cond_ample}_1 \wedge \neg \text{cond_ample}_2) \vee (\text{prob}_1 \wedge \text{cond_ample}_1) \vee (\text{prob}_2 \wedge \text{cond_ample}_2))$
 - $\sigma'_1 = \sigma_1; \text{ample}_1 = \text{false}; \text{cond_ample}_1 = \text{true}; \text{prob}_1 = \text{false}$
- $\tau' = g'_\tau : [1] \tau'_1$
 - $g'_\tau = g_\tau \wedge \neg \text{ample}_1 \wedge \neg \text{ample}_2 \wedge ((\text{prob}_2 \wedge \text{cond_ample}_2) \vee ((\neg \text{cond_ample}_1 \wedge \text{cond_ample}_2) \wedge (\neg \text{prob}_1 \wedge \neg \text{prob}_2)) \vee (\text{prob}_1 \wedge \neg \text{cond_ample}_1) \vee (\text{prob}_2 \wedge \neg \text{cond_ample}_2))$

- $\tau'_1 = \tau_1; ample_2 = true; cond_ample_2 = false$
- $\theta' = g'_\theta: [1]\theta'_1$
 - $g'_\theta = g_\theta \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((prob_1 \wedge cond_ample_1) \vee (cond_ample_1 \wedge (\neg prob_1 \wedge \neg prob_2))) \vee (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
 - $\theta'_1 = \theta_1; ample_1 = false; cond_ample_1 = false$
- $\chi' = g'_\chi: [1]\chi'_1$
 - $g'_\chi = g_\chi \wedge ample_2$
 - $\chi'_1 = \chi_1; ample_2 = false; cond_ample_2 = false$
- $\zeta' = g'_\zeta: [1]\zeta'_1$
 - $g'_\zeta = g_\zeta \wedge \neg ample_1 \wedge \neg ample_2 \wedge ((prob_1 \wedge cond_ample_1) \vee (cond_ample_1 \wedge (\neg prob_1 \wedge \neg prob_2))) \vee (prob_1 \wedge \neg cond_ample_1) \vee (prob_2 \wedge \neg cond_ample_2)$
 - $\zeta'_1 = \zeta_1; ample_1 = true; cond_ample_1 = false$

Then, we can build the associated MDP M' and start Model Checking process. Resulting Probabilistic Control Graphs PCG'_1 and PCG'_2 are presented in Figure 4.5, where dotted locations are the ones marked as *ample* and dashed ones are labelled as *cond_ample*. As we can see, this result is equivalent to the expected result described in page 37. As final remark, one should notice that in the case where no conditionally ample locations are identified, the reduced systems obtained by applying Naïve and Reachability-Aware SPOR techniques are equivalent, as despite the fact that the system modifications would be different, the location labelling would be the same in both cases.

Lemma 4.2.2. Kinds of ample sets Every ample set, from each state $s = \langle l_1, \dots, l_n, \eta' \rangle$, generated using Reachability-Aware SPOR technique belongs to one and only one of the following groups:

- a) $ample(s) = enabled(l_i)$ and $\eta' \models ample_i$. The ample set is composed by all the enabled actions from a single location l_i that is marked as *ample*.
- b) $ample(s) = enabled(l_i)$ and $\eta' \models cond_ample_i$. In this case, the ample set is composed by all the enabled actions from a single location l_i that is marked as *cond_ample*.
- c) $ample(s) = \bigcup_{l_i \in PCond} enabled(l_i)$ where $\emptyset \neq PCond = \{l_i | l_i \text{ is marked as } cond_ample \text{ and } probabilistic\}$. This ample set is composed by all the enabled actions of a set of locations that are marked as both *cond_ample* and *probabilistic*.

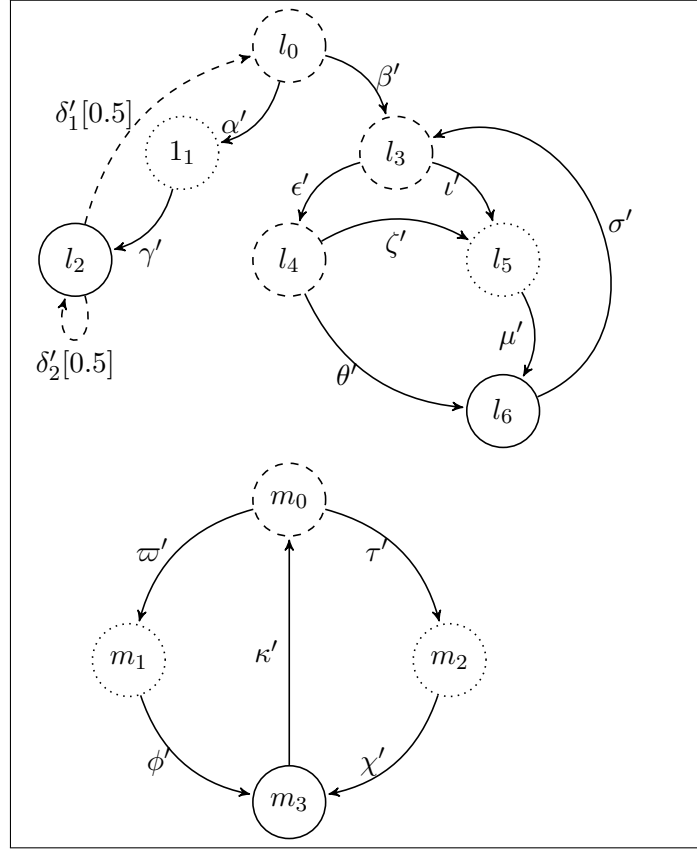


Figure 4.5: Product of Reachability-Aware SPOR technique

d) $ample(s) = Act(s)$. This case represents a full expansion.

Proof. In order to prove this lemma, we have to consider the different possible valuations η' of the propositions introduced to the guards by Reachability-Aware SPOR technique. Those valuations derive from the different labelling combinations of the locations l_i of s . This technique strengthens the guards of the different actions with three kinds of propositions, which, for the sake of simplicity, we abbreviate in the following way:

- $g_{ample-i} = h_i$ if l_i is marked as *ample*.
- $g_{cond-i} = f \wedge ((cond_ample_i \wedge prob_i) \vee (k_i \wedge q) \vee p)$ if l_i is marked as *cond_ample*.
- $g_{not-i} = f \wedge (p \vee r)$ if l_i is not marked as *ample* or *cond_ample*.

Informally, each of these strengthening propositions can be described as:

- $g_{ample-i}$ holds if l_i is marked as *ample* and its index i is the lowest index among the set of locations marked as *ample* contained in current state.

- g_{cond-i} holds if current state does not contain any location marked as *ample*, l_i is marked as *cond_ample* and at least one of the following conditions holds:
 - l_i is also marked *probabilistic*.
 - i is the lowest index among the set of locations marked as *cond_ample* contained in current state s and s does not contain any location marked as *probabilistic*.
 - At least one of the locations in current state is marked as *probabilistic* and not as *cond_ample*.
- g_{not-i} holds if current state does not contain any location marked as *ample* and at least one of the following conditions:
 - At least one of the locations in current state is marked as *probabilistic* and not as *cond_ample*.
 - None of the locations in current state is labelled as *cond_ample*.

All possible combinations of locations in a system s , along with the ample sets they generate, are:

- **There is at least one location marked as *ample*.** In this case, location l_i , where $0 < i \leq n$, is marked as *ample* and all locations l_j , such that $1 \leq j < i$ are not. That is equivalent to stating $\eta' \not\models ample_j$ for every $j < i$ and $\eta' \models ample_i$ for some $i \in [1, n]$. Then, the propositions that we have used to strengthen the guards of the actions from the different locations in current state will be determined in the following way:
 - Transition from location l_i . As this location is labelled as *ample*, its one and only outgoing transition is strengthened with proposition $g_{amp-i} = h_i = \bigwedge_{1 \leq j < i} \neg ample_j \wedge ample_i$. In this case, $\eta' \models g_{amp-i}$. Then, $enabled(l_i) \in ample(s)$.
 - Transitions from locations marked as *ample* different from l_i , if any. Every location l_k marked as *ample* has an index $k > i$ and its guards is strengthened with proposition $g_{amp-k} = h_k = \bigwedge_{1 \leq j < k} \neg ample_j \wedge ample_k$. As $i < k$ and $\eta' \models ample_i$, then $\eta' \not\models g_{amp-k}$. Therefore, for every $k \in [i + 1, n]$ where l_k is labelled as *ample*, $Act(l_k) \cap ample(s) = \emptyset$.
 - Transitions from locations marked as *cond_ample*, if any. Every location l_c marked as *con_ample* has its guards strengthened with proposition g_{cond-c} . As $\eta' \models ample_i$, then $\eta' \not\models f$, which implies $\eta' \not\models g_{cond-c}$. Therefore, for every $c \in [1, n]$ where l_c is labelled as *cond_ample*, $Act(l_c) \cap ample(s) = \emptyset$.

- Transitions from locations not marked as either *ample* nor *cond_ample*, if any. Every location l_d not marked as *ample* nor *con_ample* has its guards strengthened with proposition g_{not-d} . As $\eta' \models ample_i$, then $\eta' \not\models f$, which implies $\eta' \not\models g_{not-d}$. Therefore, for every $d \in [1, n]$ where l_d is not labelled as *ample* nor *cond_ample*, $Act(l_d) \cap ample(s) = \emptyset$.

Then, in a state s with this combination of locations, the ample set $ample(s)$ will be a singleton composed by the outgoing action from a location marked as *ample*. More formally, $ample(s) = enabled(l_i)$ which generates an ample set of type (a).

- **There is no location marked as *ample* and there is at least one location marked as *cond_ample*.** Location l_i , where $0 < i \leq n$, is marked as *cond_ample* and locations l_j , where $1 \leq j < i$, are not while any other location l_k , where $i \neq k$ and $1 \leq k \leq n$, is not marked as *ample*. Then we can state, $\eta' \not\models cond_ample_j$, $\eta' \not\models ample_k$ and $\eta' \models cond_ample_i$ which means $\eta' \models k_i \wedge f \wedge \neg r$. In this situation, depending on the reachability of probabilistic actions, we have three different cases to consider:

- **There are no probabilistic actions reachable:** then $\eta' \not\models prob_i$ for $1 \leq i \leq n$ and, consequently $\eta' \models \neg p \wedge q$. Then, the values of the strengthening propositions can be determined, as well as the composition of the ample set, in the following way:
 - * Transitions from location l_i . As this location is labelled as *cond_ample*, its outgoing transitions are strengthened with proposition g_{cond-i} . $\eta' \models f \wedge k_i \wedge q$ and, consequently $\eta' \models g_{cond-i}$. Therefore, $enabled(l_i) \in ample(s)$.
 - * Transitions from locations marked as *cond_ample* different from l_i , if any. Every location l_k marked as *cond_ample* has an index $k > i$ and its guards is strengthened with proposition g_{cond-k} . As $i < k$ and $\eta' \models cond_ample_i$, then $\eta' \not\models k_k$ which implies $\eta' \not\models g_{cond-k}$. Therefore, for every $k \in [i + 1, n]$ where l_k is labelled as *cond_ample*, $Act(l_k) \cap ample(s) = \emptyset$.
 - * Transitions from locations not marked as either *ample* nor *cond_ample*, if any. Every location l_d not marked as *ample* nor *con_ample* has its guards strengthened with proposition g_{not-d} . As $\eta' \models cond_ample_i$, then $\eta' \not\models r$, which implies $\eta' \not\models g_{not-d}$. Therefore, for every $d \in [1, n]$ where l_d is not labelled as *ample* nor *cond_ample*, $Act(l_d) \cap ample(s) = \emptyset$.

In this case, the ample set will be composed by the enabled actions from a location marked as *cond_ample*. The ample set is $ample(s) = l_i$, which belongs to group (b).

- **There are reachable probabilistic actions and condition p holds:** $\eta' \models p$ implies the existence of at least one location l_p , where $p \neq i$ and $1 \leq p \leq n$, which is marked as *probabilistic* but not as *cond_ample*. Therefore, $\eta' \models p \wedge f$ then $\forall c \in [1, n]. \eta' \models g_{cond-c} \wedge g_{not-c}$ which is equivalent to generating an ample set $ample(s) = Act(s)$ that corresponds to a full expansion and belongs to group (d).
- **There are reachable probabilistic actions and condition p fails to hold:** The case in which $\eta' \not\models p$ is equivalent to stating $\forall a. \eta' \models prob_a \rightarrow cond_ample_a$, i.e., every location marked as *probabilistic* is also marked as *cond_ample*. Then, the strengthening propositions can be determined:

- * Transitions from locations marked as *cond_ample* and *probabilistic*. Considering $\eta' \models f$, then $\forall k \in [1, n].$ s.t. $\eta' \models prob_k \wedge cond_ample_k$ then, $\eta' \models g_{cond-k}$. Consequently, $enabled(l_k) \in ample(s)$. In this case, there is at least one location labelled as *probabilistic* which is, therefore, labelled also as *cond_ample*, and whose enabled actions will belong to the ample set.
- * Transitions from locations marked as *cond_ample* but not as *probabilistic*. Considering that $\eta' \models \neg q \wedge \neg p$ then $\forall k \in [1, n]$ such that $\eta' \not\models prob_i$ then $\eta' \not\models g_{cond-k}$. The outgoing actions from those locations will be disabled, i.e., $Act(l_k) \cap ample(s) = \emptyset$.
- * Transitions from locations that are not marked as neither *cond_ample* nor *cond_ample*. Considering that $\eta' \models \neg p \wedge \neg r$ and that $\forall k \in [1, n]$ such that $\eta' \not\models ample_k \wedge cond_ample_k$ then $\eta' \not\models g_{not-k}$.

In this case, the ample set identified would contain all enabled actions from the locations that are marked both as *cond_ample* and *probabilistic*. This ample set belongs to group (c).

- **None of the locations are marked as *ample* or *cond_ample*.** In this case, $\forall i \in [1, n]. \eta' \models \neg ample_i \wedge \neg cond_ample_i$, implying that $\eta' \models f \wedge r$. Then, $\forall i \in [1, n]. \eta' \models g_{not-i}$. All outgoing actions of every location l_i have been strengthened with condition g_{not-i} , which always holds in this case. Again, the ample set generated corresponds to a full expansion, which corresponds to group (d).

□

Theorem 4.2.3. Soundness of Reachability-Aware SPOR technique

Given PCG_1, \dots, PCG_n , which generate MDP M with state space S , if we obtain PCG'_1, \dots, PCG'_n through reachability-aware SPOR technique, then the associated MDP M' , with State Space S' is such that M and M' are stutter-equivalent for quantitative and qualitative LTL properties.

Proof. In order to prove the soundness of Reachability-Aware SPOR technique, in terms of the LTL-stutter-equivalence of M and M' , we must show that the reduction performed is based on the identification of ample sets that comply with conditions C0-C3 and C4' in page 18. We must take into account the different kinds of ample sets that can be generated. Those kinds are stated in Lemma 4.2.2. Fulfilment of conditions C0-C4' can be proven by:

- **C0:** as the guard g of every action α is modified, we must show that no new deadlocks, i.e. states $s \in S'$ from which it is not possible to execute any further action, are created. This condition is guaranteed in every ample set by showing that the ample set is not empty unless the original unreduced state was :
 - a) $ample(s) = enabled(l_i)$ and $\eta' \models ample_i$. A location l_i can only be *ample* if the guard g of its single outgoing location is a tautology. Then $|enabled(l_i)| = 1$ and, consequently, $ample(s) \neq \emptyset$.
 - b) $ample(s) = enabled(l_i)$ and $\eta' \models cond_ample_i$. Again, as the disjunction of all the guards of the outgoing actions of l_i is a tautology, at least one of those actions will be enabled. Consequently, $|enabled(l_i)| > 0$ and $ample(s) \neq \emptyset$.
 - c) $ample(s) = \bigcup_{l_i \in PCond} \text{where } \emptyset \neq \emptyset \neq PCond = \{l_i | l_i \text{ is marked as } cond_ample \text{ and } probabilistic\}$. All locations $l_i \in PCond$ are marked as *cond_ample*. Then, following the same rationale as we did in the previous paragraph, we can state state there will be at least one enabled location from each of those locations. Therefore, $|ample(s)| \geq |PCond| > 0$.
 - d) $ample(s) = Act(s)$. No reduction is performed. Then, C0 is obvious.
- **C1:** we must guarantee that every reachable action β that depends on the ample set from state s , $ample(s)$, either belongs to $ample(s)$ or at least one of the actions in every finite path that leads from s to a state s' where β is enabled, belongs to $ample(s)$. We can guarantee this condition for each kind of ample set by showing that either there is no reduction or β cannot exist:
 - a) $ample(s) = enabled(l_i)$ and $\eta' \models ample_i$. As l_i is labelled as *ample*, its outgoing action cannot be dependent to any other action. Then, there is no action β dependent to $ample(s)$.

- b) $ample(s) = enabled(l_i)$ and $\eta' \models cond_ample_i$. In this case, all the outgoing actions of l_i are independent to actions in other PCG's. Therefore, an action β dependent on $ample(s)$ cannot exist.
- c) $ample(s) = \bigcup_{l_i \in PCond} \text{where } \emptyset \neq \emptyset \neq PCond = \{l_i | l_i \text{ is marked as } cond_ample \text{ and } probabilistic\}$. All locations $l_i \in PCond$ are marked as $cond_ample$. Therefore, all outgoing actions have no dependencies to other actions outside the ample set.
- d) $ample(s) = Act(s)$. No reduction is performed. Then, C1 is obvious.
- **C2-C3**: the technique uses the Sticky Action approach, $ample(s) \cap Sticky \neq \emptyset \Rightarrow ample(s) = Act(s)$, which guarantees fulfilment of condition C3', described in page 21. We can prove these conditions by showing that none of the ample sets, excepting the one equivalent to a full expansion, can contain a sticky action:
 - a) $ample(s) = enabled(l_i)$ and $\eta' \models ample_i$. As l_i is labelled as $ample$, its outgoing action cannot be labelled as $Sticky$. Then, $ample(s) \cap Sticky = \emptyset$. Then, $ample(s) \cap Sticky = \emptyset$.
 - b) $ample(s) = enabled(l_i)$ and $\eta' \models cond_ample_i$. In this case, none of the outgoing actions of l_i can be labelled as $Sticky$ and, therefore, $ample(s) \cap Sticky = \emptyset$.
 - c) $ample(s) = \bigcup_{l_i \in PCond} \text{where } \emptyset \neq \emptyset \neq PCond = \{l_i | l_i \text{ is marked as } cond_ample \text{ and } probabilistic\}$. All locations $l_i \in PCond$ are marked as $cond_ample$. Therefore, as we show in the previous paragraph, $ample(s) \cap Sticky = \emptyset$.
 - d) $ample(s) = Act(s)$. Sticky Action approach is fulfilled as the consequent holds.
 - **C4'**: we have to show that every probabilistic action β reachable from state s either belongs to $ample(s)$ or that this ample set contains at least one of the actions of every finite path in the original unreduced system that starts in s and contains β . Otherwise, either there are no more probabilistic actions reachable, or the ample set is a singleton, i.e. $|ample(s)| = 1$. We prove fulfilment of this condition by showing that each kind of ample set complies with at least one of the requirements we just enumerated:
 - a) $ample(s) = enabled(l_i)$ and $\eta' \models ample_i$. Location l_i , from PCG_i , is marked as $ample$. Then, it has one and only one outgoing transition and $|ample(s)| = 1$.

- b) $ample(s) = enabled(l_i)$ and $\eta' \models cond_ample_i$. From the proof of Lemma 4.2.2 we can see that this ample appears when location l_i , from PCG_i , is marked as $cond_ample$ and there are no probabilistic actions α reachable. As there are no probabilistic actions reachable, C4' holds.
- c) $ample(s) = \bigcup_{l_i \in PCond} enabled(l_i)$ where $\emptyset \neq PCond = \{l_i | l_i \text{ is marked as } cond_ample \text{ and } probabilistic\}$. In this case, every *probabilistic* location is also a *conditionally ample* one and all its enabled actions belong to the ample set. Therefore, all probabilistic actions reachable cannot be executed prior to the execution of one of the actions in the ample set unless they belong to it. Consequently C4' is fulfilled.
- d) $ample(s) = Act(s)$. This case corresponds to a full expansion of state s . Therefore, fulfilment of C4' is obvious.

□

4.2.5. Experiments

After defining the new static POR algorithms, it is interesting to check their efficiency. In order to obtain some empirical information about their performance in both explicit and symbolic Model Checking, we carried out a set of experiments. We started by applying Naïve and Reachability-Aware SPOR to the PRISM implementation of two classical concurrency systems: the Randomised Dining Philosophers [LR81] and the Randomised Mutual Exclusion [PZ86], both available in [PRI]. As there is no way to apply the aforementioned reduction techniques automatically yet, we applied them manually. As a result of the reduction, we generated some new PRISM code corresponding to the modifications indicated by the techniques. In both systems, Reachability-Aware SPOR did not identify any conditionally ample location, only some ample locations. Therefore, the resulting reduced systems were identical after the application of each technique, i.e., the result of applying Naïve SPOR to the Randomised Mutual Exclusion system is equivalent to the one obtained if we apply Reachability-Aware SPOR to it. We analysed some temporal properties in both resulting reduced systems and the original ones using PRISM model checker. As stutter-trace equivalence is guaranteed by both reduction algorithms, the results obtained during the property checking were the same. We include a comparison between the state spaces and transition matrices of the different reduced and unreduced systems in Tables 4.1, 4.2, 4.3 and 4.4. In order to provide a more readable comparison, we also include Figures 4.6 and 4.7.

Number of Philosophers	Number of States	Number of Transitions	Number of Nodes in MTBDD	Number of Terminal Nodes
4	9440	48656	15361	3
5	81936	477404	24148	3

Table 4.1: Unreduced Randomised Dining Philosophers System

Number of Philosophers	Number of States	Number of Transitions	Number of Nodes in MTBDD	Number of Terminal Nodes
4	8215	28324	21331	3
5	68976	277691	83165	3

Table 4.2: Reduced Randomised Dining Philosophers System

Number of Processes	Number of States	Number of Transitions	Number of Nodes in MTBDD	Number of Terminal Nodes
4	27600	136992	12355	3
6	3377344	25470144	33297	3
8	$3,9 \cdot 10^8$	$3,9 \cdot 10^9$	62871	3
10	$4,4 \cdot 10^{10}$	$5,6 \cdot 10^{11}$	101077	3

Table 4.3: Unreduced Randomised Mutual Exclusion System

We can see how, in all the cases, the number of states and transitions is reduced. It is also remarkable that the reduction percentage in those numbers seems to be constant within the same kind of system, independently from the number of processes involved. However, if we consider the number of nodes in the MTBDDs corresponding to the transition matrices, we can identify a different behaviour. In the case of the Randomised Mutual Exclusion System, we obtained a reduction that seems to increase proportionally to the number of processes in the system. In the case where there are ten processes, the reduction in the number of non-terminal nodes in the transition matrix amounts to 26%. Conversely, in the Randomised Dining Philosophers System, the number of non-terminal nodes increases. In the case where there are four philosophers, the increase is of 39% while the case for five philosopher the increase amounts to a 244%, which means that the number of nodes

Number of Processes	Number of States	Number of Transitions	Number of Nodes in MTBDD	Number of Terminal Nodes
4	21040	97360	10655	3
6	2482432	17598112	26310	3
8	$2,8 \cdot 10^8$	$2,7 \cdot 10^9$	47697	3
10	$3,1 \cdot 10^{10}$	$3,9 \cdot 10^{11}$	74816	3

Table 4.4: Reduced Randomised Mutual Exclusion System

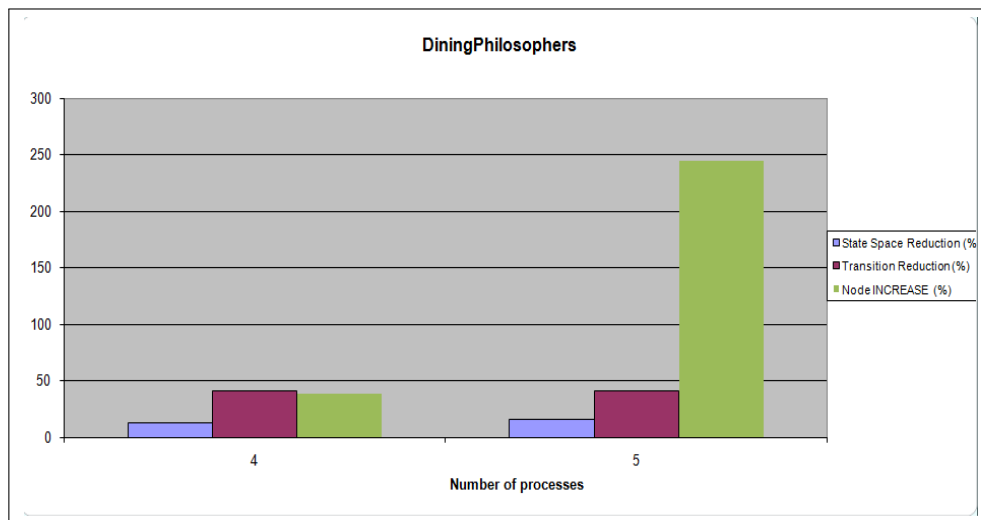


Figure 4.6: Percentage Statistics for Randomised Dining Philosophers System

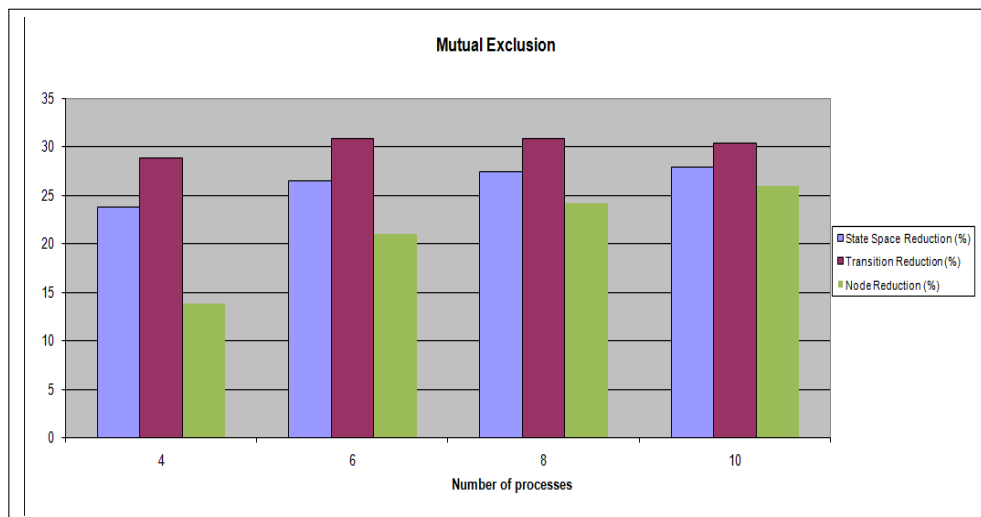


Figure 4.7: Percentage Statistics for Randomised Mutual Exclusion System

in the “reduced” system is 3.5 times bigger. We believe that such increase derives from the addition of new variables to the system. Recall that the product obtained after applying these reduction techniques is a modified system that includes some new global variables and commands to check and update them. Nevertheless, we have seen how, in some cases, the reduction in the number of transitions and states also appears in the number of non-terminal nodes corresponding to the MTBDD for the transition matrix of the system. Regarding the number of terminal nodes, it has remained constant in all experiments and systems.

As we identified a case in which the new static partial order reduction techniques successfully reduce the model of the system in combination with symbolic model checking, we wanted to check if they can be applied in combination with explicit model checking and obtain a reduction better than the one achieved with dynamic partial order reduction techniques. In order to do that, we extended LiQuor Model Checker to be able to execute both Naïve and Reachability-Aware SPOR techniques over PASM models. As an initial step, we carried out an extension aimed at creating a symbolic representation of the system. That symbolic representation contains all the information about the different probabilistic control graphs, i.e. the set of locations and transitions between them. That information is now obtained and stored in a series of data structures into LiQuor Model Checker. Once the different probabilistic control graphs are available, LiQuor can execute Naïve and Reachability-Aware SPOR techniques. Nevertheless, there is a relatively small difference between the specification of Reachability-Aware SPOR technique and its actual implementation. That difference is motivated by the high complexity to calculate whether the disjunction of a set of predicates is a tautology or not. In our implementation, that problem implies checking whether any valuation for the variables involved in the guards would always enable at least one of them. This problem is especially hard when there are variables whose domain is either infinite or relatively wide, as happens with numerical variables. Therefore, LiQuor extension only considers the disjunction of a set of guards to be a tautology if at least one of those guards is a tautology itself. We refer here to this implementation strategy as *Tautology Identification Overapproximation*.

Lemma 4.2.4. Tautology Identification Overapproximation and Reachability-Aware SPOR

If we carry out Reachability-Aware SPOR in a system in such a way that we replace the original condition for identifying a conditionally ample location by using Tautology Identification Overapproximation, the result is still valid and sound.

Proof. In order to prove this assessment we must recall, first, that the new condition that

defines whether a location can be marked as *cond_ample* is:

- A location l_i is a conditionally ample location, thus marked as *cond_ample* if and only if all its outgoing transitions, whose quantity has to be greater than one, are marked as good and at least one of their guards is a tautology.

We assume that by using Reachability-Aware SPOR over a set of probabilistic control graphs we would identify a set C of conditionally ample locations. Then, by using the overapproximation, we would identify a set $C' \subseteq C$ of conditionally ample locations. The reason that ensures the identification of a valid subset is that if a conditionally ample location l_i is identified by the overapproximation, it implies that at least one of the guards g of its outgoing locations is a tautology. Therefore, the original condition, which requires that the disjunction of all guards must be a tautology, holds as any proposition constructed by adding a disjunction to a tautology is a tautology itself. The consequence of finding a possibly smaller set of conditionally ample locations is a decrease in the efficiency of the reduction technique, as there will be less ample sets that do not correspond to a full expansion. However, as stated in Consideration1 in page 33, omitting ample locations still gives a sound solution as it resumes to skipping reduced ample sets. \square

Once LiQuor was successfully extended to perform the new static partial order techniques introduced in this document, we chose the PROBMELA versions of the Randomised Dining Philosophers and the Leader Election algorithm as candidate systems to be reduced. As LiQuor, by now, only allows the model checking of qualitative and quantitative LTL properties, the experiments only ran Reachability-Aware SPOR technique whose reduction is at least as good as the one obtained with Naïve SPOR. After we run the experiments, we noticed that in the case of the Randomised Dining Philosophers no reduction was achieved. The reason why there was no reduction derives from the fact that no ample or conditionally ample locations were identified. Those kind of locations could not be identified due to the fact that all actions in that system accessed (reading, updating or both) global variables that were also accessed by more than one process. Therefore, the resulting modified system was equivalent to the original one in terms of the size of the state space and transitions and in the properties that it models, provided that those properties do not refer the new variables introduced by the reduction technique. In the case of the Leader Election System, there where two ample locations identified in every probabilistic control graph, from a total of 25 locations in each of them. In order to provide an efficiency comparison with the dynamic partial order reduction techniques already implemented in LiQuor, we also ran them over the Leader Election System. The results we obtained are presented in Table 4.5 and Figure 4.8 shows a percentage comparison diagram.

Number of Processes (Reduction Technique)	Number of States	Number of Transitions	Time for State Space Generation (s)
2 (not reduced)	3058	5626	1
2 (SPOR)	2743	4934	1
2 (DPOR)	2304	3009	1, 5
3 (not reduced)	60296	149910	52
3 (SPOR)	55803	133326	53, 3
3 (DPOR)	52820	95199	53, 9
4 (not reduced)	3068194	9126120	909, 7
4 (SPOR)	1391973	3927710	1390
4 (DPOR)	1265689	1938194	2809

Table 4.5: Reduced Leader Election System

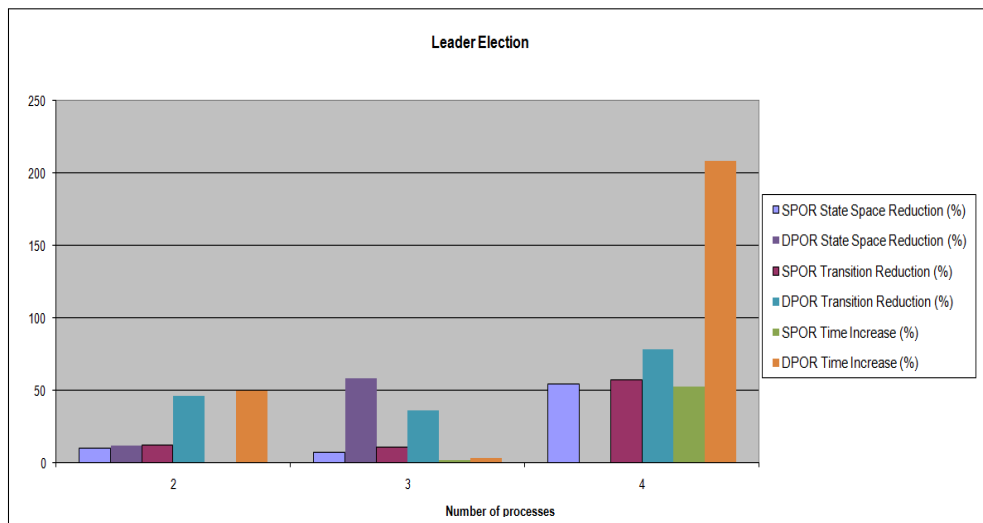


Figure 4.8: Percentage Comparison between SPOR and DPOR

We can see how even though we can obtain some reduction in both the number of states and transitions, which amount to a maximum of 54% and 56% respectively. In this case, that reduction is worse than the one obtained by using dynamic partial order reduction techniques, whose maximum value reached a 58% reduction in the number of states and a 78% in the number of transitions. The only improvement that is perceived is a smaller increase in the time required to generate the state space of the system by using static POR, compared with the increase originated from the use of dynamic POR.

From this results we obtain some evidence that supports an intuition that we had prior to the comparison allowed by the different experiments. That intuition states that static partial order techniques are more suitable for being used in combination with symbolic model checking that to be applied to systems where explicit model checking is used as verification technique. One possible motivation for this is the amount of information available in the moment when the ample sets are defined. In the static POR, all the information is extracted from the configuration of the system on the basis of the different locations and transitions between them that appear in the probabilistic control graphs of the processes in the system. In dynamic POR techniques, that information is also available, together with some other that can be gathered during the state space generation. That extra amount of information dynamically obtained can be gathered when explicit model checking is used, but cannot in the case of symbolic model checking, which then only benefits from static POR techniques.

5

Conclusion and Future Lines of Work

During the development of this Master Thesis, we have carried out a brief analysis over the state-of-the-art of the partial order reduction techniques for the formal verification of qualitative and quantitative temporal properties of probabilistic systems via model checking. We identified the lack of static techniques to allow the partial order reduction of probabilistic and non-deterministic models, like the ones represented as Markov Decision Processes (MDP). On the basis of an already existing static partial order technique for non-probabilistic systems, we developed two new completely static techniques that allow the reduction of system models represented as MDPs.

The first of the techniques, namely Naïve SPOR, can be applied when the properties to be checked can be specified in the modal logic PCTL* and are stutter-invariant, i.e. expressed in PCTL^*_{\circ} where the *next step* operator \circ is not used. This technique guarantees ample conditions C0-C4, described in page 18, which are necessary and sufficient to ensure stutter-equivalence between the original and reduced models. The second technique, namely Reachability-Aware SPOR, cannot be applied to every quantitative PTCL^*_{\circ} property, but the quantitative ones that can be written in LTL_{\circ} . As stated in [BDG06], those properties are PTCL^*_{\circ} formulas where all the state subformulas are propositional, i.e. they do not contain the probabilistic operator. This technique guarantees that the reduced system is built by constructing ample set that fulfil conditions C0-C3 and C4', which is weaker than C4. Therefore, this technique can produce some reductions, mentioned in page 36 that Naïve SPOR cannot. As both techniques guarantee conditions C0-C3, they can be applied for the reduction of non-probabilistic systems, but we also gave theoretical evidence which support that the reduction achieved by using Reachability-Aware SPOR can be bigger.

During the experimentation phase, we selected some classical concurrency models and applied both reduction techniques on them. Some of those systems were modelled in SPIN, see [Hol03], thus applying symbolic model checking to them, whereas others were written in PROBMELA, see [BCG04]. In this latter case, we applied explicit model checking to the models and, in order to be able to use our reduction techniques, we extended the

functionality provided by LiQuor model checker, see [CC06]. As result of the experimentation phase, we realized that symbolic model checking benefits more from static POR than explicit model checking, as the latter also allows dynamic POR which produced better reductions in the cases used. We believe that the rationale for that derives from the fact that explicit model checking can dynamically provide some information about the system which is not statically accessible. Dynamic POR techniques can use information obtained both statically and dynamically. Therefore, a bigger amount of information can lead to better reductions. Symbolic model checking can only benefit from static POR, making this a better match than the combination of static POR and explicit model checking. However, we provide evidence that shows that a static POR reduction of the system is achievable even if the use of explicit model checking is intended.

As we report in the previous section, in none of the cases Reachability-Aware SPOR technique was able to identify a conditionally ample location. Therefore, the reduced model achieved by using both techniques was equivalent. Nevertheless, that kind of locations theoretically exist and, in order to be able to provide a better comparison between the two new SPOR techniques, we intend to find a system that contains at least one of them as future line of work. Another possible future line of work consists in the improvement of the identification of the dependency relation between actions. Currently, we calculate the dependency relation using the overapproximation described in page 34, extracted from [BK08]. Moreover, that dependency relation D is later expanded by our SPOR techniques in order to contain all the pairs of actions $(\alpha, \beta) \in Act_i \times Act_j$ such that the pair of actions $(\delta, \gamma) \in Act_i \times Act_j$ already belonged to D . That implies that two actions α and β are considered dependent if the dependency relation overapproximation considers as dependent two actions δ and γ such that, α and δ belong to the same probabilistic control graph PCG_i and, similarly, β and γ belong to PCG_j . Therefore, two actions can be dependent even if they do not read or write the same variable. Then, the final dependency relation used by the reduction techniques is constant during all the reduction process and disregards the reachability of the actions that generated the new dependencies. For instance, considering the example just provided, in order to build the ample set from a state s , (α, β) would belong to D , and considered dependent, even if neither δ nor γ are reachable from s . This improvement would require the definition, implementation and use of a dynamic dependency relation that varies from state to state.

Finally, another future line of work would be aimed at improving the end component identification strategy. As we state in Chapter 3, the Sticky Set POR technique requires that every end component in the reduced system should contain at least one sticky action. Currently, we guarantee this condition by using the Naïve Sticky Set Identification Algorithm, which deals with cycles in the different probabilistic control graphs instead of end components in the state space. That new algorithm would be aimed at guaranteeing that every end component, then not necessarily every cycle in the control graphs, would contain at least one sticky action. Therefore, as the generated Sticky Set would be smaller, the reduction achieved by our SPOR techniques could be improved.

List of Figures

2.1. MDP digraph for the stochastic counter	10
2.2. Probabilistic Control Graph for the stochastic counter	11
2.3. Individual Probabilistic Control Graphs for the 2-counters system	12
2.4. Crossproduct Probabilistic Control Graph for the 2-counters system	13
2.5. Fully interleaved subgraph of the 2-counters system	15
2.6. Single interleaving subgraph of the 2-counters system	16
3.1. Naïve Sticky Set Identification Algorithm	23
3.2. Sticky Actions in the individual Control Graphs of 2-counters system	24
3.3. Crossproduct Probabilistic Control Graph for the 2-counters showing Sticky Set	25
3.4. Statistics for four Dining Philosophers	26
3.5. Statistics for five Dining Philosophers	27
3.6. Statistics for a Leader Election between 3 processes	28
3.7. Statistics for a Leader Election among 4 processes	28
4.1. Program Graphs of the iterative counters	31
4.2. Modified Program Graphs of the iterative counters	32
4.3. Comparison between Naïve SPOR (above) and non-probabilistic SPOR (below)	37
4.4. Yet another shortcoming of Naïve SPOR	38
4.5. Product of Reachability-Aware SPOR technique	45
4.6. Percentage Statistics for Randomised Dining Philosophers System	53
4.7. Percentage Statistics for Randomised Mutual Exclusion System	53
4.8. Percentage Comparison between SPOR and DPOR	56

List of Tables

- 4.1. Unreduced Randomised Dining Philosophers System 52
- 4.2. Reduced Randomised Dining Philosophers System 52
- 4.3. Unreduced Randomised Mutual Exclusion System 52
- 4.4. Reduced Randomised Mutual Exclusion System 53
- 4.5. Reduced Leader Election System 56

Bibliography

- [BCG04] Christel Baier, Frank Ciesinski, and Marcus Größer. ProbMela: a modeling language for communicating probabilistic processes, 2004.
- [BCHG⁺97] Christel Baier, Edmund M. Clarke, Vassili Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. Symbolic model checking for probabilistic processes. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming, ICALP '97*, pages 430–440, London, UK, 1997. Springer-Verlag.
- [BCM⁺90] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10 20 states and beyond, 1990.
- [BDG06] Christel Baier, Pedro D'Argenio, and Marcus Groesser. Partial order reduction for probabilistic branching time. *Electronic Notes in Theoretical Computer Science*, 153(2):97–116, 2006. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005).
- [BK08] C. Baier and J-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
- [Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [CC06] F. Ciesinski and C. Baier. Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In *Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*. IEEE CS Press, 2006.
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [CGC04] C. Baier, M. Grösser, and F. Ciesinski. Partial order reductions for probabilistic systems. In *Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04)*, 2004.
- [Cie11] F. Ciesinski. *High-Level Modelling and Efficient Analysis of Randomized Protocols*. PhD thesis, Technische Universität Dresden, 2011.

- [Cla08] Edmund Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2008.
- [Din07] Monica Dinculescu. Probabilistic temporal logic or: With what probability will the swedish chef bork the meatballs?, 2007.
- [DN04] Pedro R. D’Argenio and Peter Niebert. Partial order reduction on concurrent probabilistic programs. In *Proceedings of the The Quantitative Evaluation of Systems, First International Conference*, pages 240–249, Washington, DC, USA, 2004. IEEE Computer Society.
- [EH82] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC ’82, pages 169–180, New York, NY, USA, 1982. ACM.
- [Eme08] E. Allen Emerson. 25 years of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, chapter The Beginning of Model Checking: A Personal Perspective, pages 27–45. Springer-Verlag, Berlin, Heidelberg, 2008.
- [FG05] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL ’05)*, 2005.
- [FMY97] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10:149–169, 1997. 10.1023/A:1008647823331.
- [FS07] Lars-Ake Fredlund and Hans Svensson. Mcerlang: a model checker for a distributed functional programming language. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP ’07, pages 125–136, New York, NY, USA, 2007. ACM.
- [GKPP99] Rob Gerth, Ruurd Kuiper, Doron Peled, and Wojciech Penczek. A partial order approach to branching time logic model checking. *Information and Computation*, 150(2):132 – 152, 1999.
- [God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

- [HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994. 10.1007/BF01211866.
- [HKPM97] G. Huer, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant: A tutorial. Technical report, INRIA, 1997.
- [Hol03] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.
- [KLM⁺98] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1998.
- [KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: Probabilistic symbolic model checker. In Tony Field, Peter Harrison, Jeremy Bradley, and Uli Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*, pages 113–140. Springer Berlin / Heidelberg, 2002.
- [LR81] D. Lehmann and M. Rabin. On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract). In *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*, pages 133–138, 1981.
- [Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in *Lecture Notes in Computer Science*. Springer – Berlin, 1994.
- [Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer Berlin / Heidelberg, 1993.
- [Pel98] D. Peled. Ten years of partial order reduction. In *Computer Aided Verification (CAV'98)*, 1998.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.
- [PRI] PRISM. Prism case studies. <http://www.prismmodelchecker.org/casestudies/index.php>.

-
- [PRO] *PROMELA Manual Pages*.
- [PZ86] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.
- [Val89] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989, Bonn, Germany; Supplement*, pages 1–22, 1989. NewsletterInfo: 33,39.
- [Var98] Moshe Vardi. Sometimes and not never re-revisited: on branching versus linear time. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055612.

Declaration

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Affidavit

I herewith declare, in lieu of oath, that I have prepared this paper on my own, using only the materials (devices) mentioned. Ideas taken, directly or indirectly, from other sources, are identified as such.

29th June 2011

Alvaro