Dissertation, submitted to the Carl von Ossietzky Universität Oldenburg
in partial fulfillment of the requirements for the degree of Doktor-Ingenieur

Unmasking fault tolerance:
Quantifying deterministic recovery dynamics
in probabilistic environments

by Nils Henning Müllner

	Examining board:	
Chair	**Prof. Dr. Martin Fränzle**	
	Carl von Ossietzky Universität Oldenburg, Germany	
First Supervisor	**Prof. Dr.-Ing. Oliver Theel**	
	Carl von Ossietzky Universität Oldenburg, Germany	
Second Supervisor	**Prof. Dr. Ir. Joost-Pieter Katoen**	
	RWTH Aachen, Germany	
	Universiteit Twente, Netherlands	
Faculty Member	**PD Dr. Elke Wilkeit**	
	Carl von Ossietzky Universität Oldenburg, Germany	
Day of Submission	Monday, 30 September 2013	
Day of Defense	Wednesday, 26 February 2014	

Faculty II - Department of Computer Science
Oldenburg, Germany

"To infinity. . . and beyond!"
Cpt. Buzz Lightyear

Abstract (English)

The present thesis focuses on distributed systems operating under probabilistic influences like faults. How well can such systems provide their service under the effects of faults? How well can they recover from faults? The present thesis introduces with *limiting window availability* a suitable measure to answer such questions, and presents a method for its computation. For the computation, the transition models of the systems are constructed, which are exponential in the size of the constituting system models. This is known as state space explosion. Combining decomposition and lumping — methods for state space reduction from the domain of model checking — allows to dampen the state space explosion.

Kurzzusammenfassung (Deutsch)

Die vorliegende Arbeit betrachtet verteilte Systeme, welche unter wahrscheinlichkeitstheoretischen Einflüssen, beispielsweise Fehlern, operieren. Wie gut können solche Systeme unter den Auswirkungen von Fehlern ihren Dienst erbringen? Wie gut können sie sich von Fehlern erholen? Die vorliegende Arbeit stellt mit der *limiting window availability* ein geeignetes Maß zur Beantwortung dieser Fragen vor und präsentiert eine Methode, um es zu berechnen. Zur Berechnung werden die Transitionsmodelle der Systeme konstruiert, welche exponentiell in der Größe der zugrundeliegenden Systemmodelle sind. Dies ist auch bekannt als Zustandsraumexplosion. Die Kombination von Dekomposition und Lumping — Methoden zur Zustandsraumreduktion aus dem Bereich Modellprüfung — erlaubt es, die Zustandsraumexplosion zu dämpfen.

Declaration of authorship

The present dissertation was composed by myself. Its content has not been published as Diploma- or similar examination (except where cited accordingly) and the employed resources are completely declared. The present dissertation was composed according to the guidelines for research integrity and good scientific practice by the DFG. The author did neither consume any stimulants other than coffee for the time of being a PhD student, nor did he task a commercial consulting or placing service.

Eigenständigkeitserklärung

Die vorliegende Dissertation wurde selbstständig verfasst. Ihr Inhalt wurde nicht für eine Diplom- oder ähnliche Prüfungsarbeit verwendet (soweit nicht anders angegeben) und die verwendeten Hilfsmittel sind vollständig angegeben. Die vorliegende Dissertation wurde den Regeln guter wissenschaftlicher Praxis entsprechend der DFG-Richtlinien verfasst. Der Autor hat als Doktorand weder Stimulanzien außer Kaffee konsumiert, noch einen kommerziellen Beratungs- oder Vermittlungsdienst in Anspruch genommen.

Nils Henning Müllner

Acknowledgments

This thesis is the result of a trace, a sequence of singular probabilistic events. The outcomes of the events were (most of the times) to my advantage, for which I owe to many people who I would like to thank here. Without each of them, the result would have been a different one. First of all, I owe thanks to my doctor-father Professor Oliver Theel for taking me under his wing, for his endless patience and for letting me pursue the topic so freely. I met Professor Joost-Pieter Katoen during a MoVeS meeting at TU Delft for the first time and was intrigued by his competence in model checking. I thank you for becoming my second supervisor, for sharing your knowledge with me and for fruitful discussions along the way. Professor Martin Fränzle employed me after my stipend ended. I am grateful that I could disseminate and apply some of my results under his guidance within the MoVeS project and in turn benefit from a great deal of the MoVeS experience to improve this thesis. I thank you for your continuous availability, for sharing your incredible amount of expertise and for your calming yet distinct way of working things out. Last but definitely not least I thank Dr. Elke Wilkeit for introducing me to the scientific method in the first place. I still remember my first steps writing my "individuelles Projekt" under her guidance in 2006 and how it sparked my interest in distributed computing.

While Professor Theel is my doctor-father, Drs.-Ing. Jens Oehlerking and Abhishek Dhama, Professor Andreas Schäfer and Dr. Sebastian Gerwinn acted as my doctor-older-brothers who I could ask any question at any time. Your support is invaluable. I thank PD Dr. Sibylle Fröschle, Ulrich Hobelmann and Annika Schwindt for their proof-reading and Professors Sandeep Kulkarni and Sèbastien Tixeuil and Dr. Vassilios Mertsiotakis for sharing their expertise. I would also like to take the opportunity to commemorate Professor Mieso Denko from the University of Guelph, Canada, who passed away unexpectedly on 27 April 2010. I met Professor Denko at the Symposium on UbiCom Frontiers in Brisbane in 2009. He gave me a great portion of motivation by accepting my second paper. Another big boost of motivation came in Japan in 2012. I thank Professors Makoto Takizawa and Leonard Barolli for awarding my fourth paper with the AINA Best Paper Award that year.

One continuous factor for which I am very grateful is the friendly work environment. I thank my colleagues Christian Ellen, Eike Möhlmann, Oday Jubran, Andreas Eggers, Dr. Kinga Lipskoch, Sven Linker, Hendrik Radke, Felix Oppermann, Robert Schadek, Eckard Böde, Philip Rehkop, Brian Clark, Christoph Etzien, Dr. Stephanie Kemper, Markus Oertel, Thomas Peikenkamp, Axel Reimer, Dr. Michael Siegel, Sven Sieverding, Daniel Sojka and Raphael Weber. I thank Pietu Pojahlainen from the University of Helsinki for inviting me to his winter school as lecturer in 2008. I hope we meet again.

Last but not least I owe my deepest gratitude to my parents Ingeborg and Helmut, my sister Nina and my girlfriend Katrin Gese. Without your ongoing encouragement this would not have been possible.

Thank you all for being part of this amazing trace!

List of publications

Below are listed the peer-reviewed publications that were published during the writing of this thesis between 2008 and 2013. Parts of the present dissertation are based on these references. The contributions are listed in descending order of publication date.

[Müllner et al., 2008] Müllner, N., Dhama, A., and Theel, O. (2008). Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation. In *Proceedings of the 41st Annual Symposium on Simulation (AnSS2008)*, pages 183 – 192, Ottawa, ON, Canada. IEEE Computer Society Press

[Müllner et al., 2009] Müllner, N., Dhama, A., and Theel, O. (2009). Deriving a Good Trade-off Between System Availability and Time Redundancy. In *Proceedings of the Symposia and Workshops on Ubiquitous, Automatic and Trusted Computing*, number E3737 in Track "International Symposium on UbiCom Frontiers - Innovative Research, Systems and Technologies (Ufirst-09)", pages 61 – 67, Brisbane, QLD, Australia. IEEE Computer Society Press

[Müllner and Theel, 2011] Müllner, N. and Theel, O. (2011). The Degree of Masking Fault Tolerance vs. Temporal Redundancy. In *Proceedings of the 25th IEEE Workshops of the International Conference on Advanced Information Networking and Applications (WAINA2011)*, Track "The Seventh International Symposium on Frontiers of Information Systems and Network Applications (FINA2011)", pages 21 – 28, Biopolis, Singapore. IEEE Computer Society Press

[Müllner et al., 2012] Müllner, N., Theel, O., and Fränzle, M. (2012). Combining Decomposition and Reduction for State Space Analysis of a Self-Stabilizing System. In *Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA2012)*, pages 936 – 943, Fukuoka-shi, Fukuoka, Japan. IEEE Computer Society Press. Best Paper Award

[Müllner et al., 2013] Müllner, N., Theel, O., and Fränzle, M. (2013). Combining Decomposition and Reduction for the State Space Analysis of Self-Stabilizing Systems. In *Journal of Computer and System Sciences (JCSS)*, volume 79, pages 1113 – 1125. Elsevier Science Publishers B. V. The paper is an extended version of a publication with the same title

[Kamgarpour et al., 2013] Kamgarpour, M., Ellen, C., Soudjani, S. E. Z., Gerwinn, S., Mathieux, J. L., Müllner, N., Abate, A., Callaway, D. S., Fränzle, M., and Lygeros, J. (2013). Modeling Options for Demand Side Participation of Thermostatically Controlled Loads. In *Proceedings of the IREP Symposium-Bulk Power System Dynamics and Control -IX (IREP), August 25-30, 2013, Rethymnon, Greece*

[Müllner et al., 2014a] Müllner, N., Theel, O., and Fränzle, M. (2014a). Combining Decomposition and Lumping to Evaluate Semi-hierarchical Systems. In *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA2014)*

[Müllner et al., 2014b] Müllner, N., Theel, O., and Fränzle, M. (2014b). Composing Thermostatically Controlled Loads to Determine the Reliability against Blackouts. In *Proceedings of the 10th International Symposium on Frontiers of Information Systems and Network Applications (FINA2014)*

Contents

1. Introduction

Fault tolerance generally is the ability of a system to fulfill a desired task even in the presence of faults. Over the past decades, this ability has been discussed for a variety of systems and a wide range of application scenarios. This thesis focuses on distributed systems with the ability to recover from the effects of faults. These systems contain processes that cooperate to allow for recovery.

Quantifying fault tolerance

The goal of this thesis is quantifying fault tolerance properties of distributed system with deterministic dynamics and a probabilistically faulty environment to measure the recovery. To achieve this, a *deterministic* system is put into an *probabilistic* environment and it is observed, how well it *recovers*.

For instance, assume a set of data-gathering interconnected buoys in the ocean, transmitting their data to one central buoy that can upload the collected data in real time via a satellite uplink. This setup provides the distributed system and its deterministic dynamics. Further, assume the communication between the buoys to be prone to faults, thus providing a probabilistic faulty environment. Property desired for evaluation can be the *timely availability* and *consistency* of the data measured. The goal of this thesis is to develop concepts for quantifying such properties and to develop methods for their computation.

Evaluating deterministic system dynamics under probabilistic influence

The first part of the thesis focuses on fault tolerance in general to derive a suitable measure in the context of this thesis. The second part reasons about methods to compute the desired method. One suitable formalism in this context are Markov models. When the dynamics of a deterministic distributed system is combined with probabilistic influence, a probabilistic transition model can be constructed with which the desired properties can be evaluated. The systems focused in this thesis have a discrete state space and execute in discrete computation steps. Therefore, discrete time Markov chains (DTMC) are selected as transition model.

Both fault tolerance and probabilistic reasoning with Markov chains are attractive research topics. Liskov was awarded with the A. M. Turing Award for her contributions to "the

practical and theoretical foundations of [...] system design, especially related to [...] fault tolerance and distributed computing"[1] in 2008, showing the importance in determining the fault tolerance properties of distributed systems. Hillston was awarded with the BCS/CPHC Distinguished Dissertation award in 1995 for her PhD thesis on "Compositional Markovian Modelling Using a Process Algebra" [Hillston, 1995]. The *performance evaluation process algebra* (PEPA) she developed contributed to the theoretical foundation that is exploited in this thesis. Their work being highly awarded by the scientific community reflects the significance of the topic.

The primary objectives

The development of novel measure called *limiting window availability* (LWA) to quantify the probabilistic aspects of system recovery is the primary objective. LWA is practically a probability sequence over first-hitting-times, regarding the first time a system recovers to a legal state. Furthermore, a method to compute LWA based on DTMCs is developed.

The challenge in the approach lies in the DTMC being exponential in the size of the constituting system, an effect commonly known as *state space explosion*. A technique to minimize the size of a DTMC by pruning information that is not relevant to the computation of the desired measure is known as *lumping*. In order to evade to necessity to construct the full product chain before lumping can be applied, constructing the much smaller Markov chains of the subsystems — known as *marginals* or *sub-Markov chains* — provides the required leverage. Lumping can then be applied on the sub-Markov chains which are sequentially composed afterwards. This method has been successfully applied for mutually independent systems as discussed in paragraph "Related work" on page 78. On the contrary, this thesis focuses on *cooperating* processes that are mutually depending. To fulfill the goal of developing a method to compute the LWA, it is necessary to adapt decomposition, lumping and composition to the context of such systems.

But are systems — and their respective transition models — always *too large* to be analyzed? The average system size[2] grew over the past six decades, irrespective of whether the system to by analyzed is a hard- or software system. Vaandrager and Rozenberg [Rozenberg and Vaandrager, 1996] expect a doubling of code-size for software systems every two years, just like Moore's law [Moore, 1965] proclaims a similar trend for the hardware domain. Conclusively, Wirth [Wirth, 1995] assumes that software complexity increases at a slightly higher pace than hardware complexity.

While the complexity of computing the limiting window availability is exponentially proportional to the system size, the systems grow at an exponential pace themselves. Hence, it is highly desirable to reason about possibilities to at least *dampen* the effects of the state space explosion.

Limitations of the approach

The quality of the results computed by the proposed methods depends on the quality of the input data. This input data consists of deterministic system dynamics and a probabilistic environment. While the deterministic system model is assumed to be realistic in this thesis, the quality of the result hinges on the quality of the probabilistic environment. The

---

[1]As quoted in the corresponding notification by the ACM.

[2]Here, the average system size is accounted for by the number of the components it consists of.

following example of rare natural events shows that it can be challenging to precisely account for probabilistic environmental events.

Freak waves are extraordinary high waves at sea of rare occurrence. They were perceived as purely fictional until New Year's Eve 1995, when the Norwegian oil rig Draupner-E measured a wave of 26 meters altitude. After a second incident, when the ship Queen Elizabeth II reported a freak wave on her passage from Cherbourg to New York on September 11ᵗʰ 1995, scientists began to develop a probability model for the occurrence of freak waves. While initial approaches assumed that the occurrence of freak waves is based on the Rayleigh distribution [Kharif and Pelinovsky, 2003], further reported incidents insinuate that freak waves are far more common [Shemer and Sergeeva, 2009].

Since the occurrence of transient faults is — like freak waves — often based on a probability distribution, determining that distribution precisely[3] is challenging. The analysis with the concepts developed in this thesis is precise, but only as good as the input data. On the bright side, the analysis can be easily reevaluated when more precise data is available.

1.1 Practical application scenarios

This section briefly discusses practical application scenarios and how quantifying limiting window availability could benefit to their particular context.

Power grids

Consumers in a power grid demand energy when they want to use an electrical appliance. Their demand is probabilistic. When too many consumers simultaneously increase or decrease their demand simultaneously, the power grid blacks out. Energy suppliers have the task to plan the energy demand ahead. They estimate the energy demand in the future, adding a small amount of ventable excess energy to minimize the probability that the system blacks out. The amount of excess energy must ensure that the risk of black out does not exceed a certain limit. At the same time, the amount of excess energy is to be minimized in order to be able to offer competitive prices. The goal in this scenario is to determine, if a specified amount of excess energy suffices to uphold a desired probability that a black out does not occur. The LWA in this case allows to determine the duration of a black out. It answers the question: In case of a black out, how long does it take for the system to become operable again, such that every household is sufficiently supplied with energy?

Sensor networks

In sensor networks, the components of the distributed system are autonomous sensor motes gathering environmental data like temperature or humidity. Notable field studies are the vineyard project [Burrell et al., 2004], which also provides a discussion about realistic fault assumptions, Duck Island [Mainwaring et al., 2002], focusing on the stationary monitoring of a bird habitat, and "ZebraNet" [Juang et al., 2002], which concerns the mobile monitoring of a flock of zebras. A sensor network should provide the status of the motes with minimal message loss. Increasing the update frequency promotes message congestion and loss. The first-hitting-time here is the availability of sensor data for each sensor mote. By analyzing the availability of data in relation to the update frequency, the *sweet spot* between both can be determined.

---

[3]One exemplary discussion about this topic demonstrating its practical relevance is provided by Schroeder et al. [Schroeder et al., 2009]. The phenomenon of such influence is sometimes also referred to as *soft errors*.

1.2 Hypothesis

We assume i) a distributed system with deterministic dynamics and the ability to recover from the effects of transient faults[4], and ii) a probabilistic environment to influence the distributed system. Computing fault tolerance properties like recovery dynamics is a highly desirable task. A novel fault tolerance measure accounting for the effectiveness of recovery called *limiting window availability* is introduced to accomplish this task. It has to be presented in the light of the state of the art. The expected challenge in computing the limiting window availability lies in i) the processes of the underlying system being mutually depending, and ii) the size of the transition model being exponential in the size of the system. In order to compute the recovery dynamics of even *large* systems, approaches to *dampen* the state space explosion are an important objective.

1.3 Thesis structure

Chapter 2 introduces the system model used in this thesis. Chapter 3 provides a fault tolerance taxonomy suitable to discuss the recovery of distributed systems. Chapter 4 proposes *limiting window availability* as a novel fault tolerance measure to account for the recovery in distributed systems in the context of this thesis. Chapters 5 and 6 concern dampening the state space explosion in the light of mutually depending processes. The application of concepts and methods that are proposed in this thesis is shown in chapter 7. Chapter 8 concludes the key points of this thesis and provides a rich set of promising future directions in this topic.

---

[4]In the context of this thesis, the effects of faults are errors and failures. The threat taxonomy is explained in detail in section 2.2.1.

2. System, environment and transition model

This chapter introduces a general system model to discuss the analysis of fault tolerance properties. It further introduces models for environmental influence like faults or distributed scheduling. From the context of this thesis, the goal is to exploit the information of the system and environment models to construct a DTMC to determine the system's fault tolerance.

2.1 System model

The components of a distributed **system** $\mathfrak{S}$ collaborate — thereby communicating — to achieve a common goal. The components are referred to as set of **processes**. A set of processes is labeled $\Pi = \{\pi_1, \ldots, \pi_n\}$. Two processes $\pi_i, \pi_j \in \Pi$ sharing a **communication channel** labeled $e_{i,j}$ are called *neighbors*. Processes and communication channels together are also referred to as *system topology*. Processes execute an **algorithm** labeled $\mathfrak{A}$ to achieve a common goal. The processes store only that part of the algorithm that they require to execute correctly, referred to as *sub-algorithm*. An example is provided in figure 4.6 on page 57. Each process contains two memory partitions, one *static* partition containing the particular sub-algorithm and one *dynamic* partition for the process variables. The process variables are stored in **registers** R. To assume that the algorithm is in a static partition while the process variables are stored in volatile memory is motivated in paragraph "Immunity of algorithm and scheduler to faults" on page 8. The algorithms introduced in this thesis require one register per process. The register of process π_i is labeled R_i.

Definition 2.1 (System model).
A distributed system $\mathfrak{S}$ is a tuple $\mathfrak{S} = \{\Pi, E, \mathfrak{A}\}$ comprising

- *a finite, non-empty set of processes $\Pi = \{\pi_1, \ldots, \pi_n\}$,*

- *a finite, non-empty set of edges E connecting the processes $E = \{e_{i,j}, \ldots\}$ such that*

  - *$e_{i,j}$ connects processes π_i and π_j,*

  - *every edge is bidirectional,*

  - *and each process is reachable from any other process via a finite number of processes, and*

- *an algorithm $\mathfrak{A}$.*

We assume that the number of processes $|\Pi|$ is larger than 1 or else the system would not be distributed. Hence, E is non-empty. Communication among the processes is commonly realized via either message passing or shared memory access as discussed for instance in [Lamport, 1986a, Lamport, 1986b, Afek et al., 1997] and [Dolev, 2000, p.73]. In this thesis a register R_i is considered to be write- and read-accessible by its own process π_i and read-accessible by all neighbors of its process $\pi_j : e_{i,j} \in E$.

Definition 2.2 (System state).
The system state $s_t = \langle R_1, \ldots, R_n \rangle$ is the snapshot over all registers at time t.

Definition 2.3 (State space).
The state space $\mathcal{S}$ is the set of all possible states of the system.

The state space $\mathcal{S}$ contains all possible permutations of register domains. The set of initial states $\mathcal{S}_0$ is a non-empty subset of $\mathcal{S}$ with regards to algorithm $\mathfrak{A}$. In the examples provided in this thesis, the set of initial states always coincides with the state space. For instance, assume two traffic lights controlling a crossing. Then, the state space contains any permutation of two values of the set $\{green, yellow, red\}$: $\mathcal{S} = \{\langle green, green \rangle, \ldots, \langle red, red \rangle, \}$. The transitions between the states are controlled by the algorithm.

Algorithm

An algorithm $\mathfrak{A}$ is a set of guarded commands. A guarded command is an atomic command guarded by a Boolean expression. If the expression evaluated to *true*, the guard is enabled and the command can be executed within one, in this thesis atomic, computation step. Each guarded command is a triple

$$a_k : g_k \rightarrow c_k \tag{2.1}$$

with a unique *label* a_k, a *guard* g_k and a *command* c_k. The label a_k is required to address the commands. A guard g_k is a Boolean expression over read-accessible registers. A guard is *enabled* when it evaluates to $true$. When selected to execute a computation step by the scheduler, that is introduced in the next section, a process executes one command for which its guard is *enabled*. For sake of clarity, the algorithms presented in this thesis are *deterministic*, meaning that always exactly one guard per process is enabled.

Restricting communication via guards

As mentioned in the introduction, the decomposition of systems is later necessary to apply the reduction method of lumping on the transition models of the subsystems. It is noticeable in that context that the bidirectional communication channels allow faults to propagate generally in any direction. Yet, the algorithms possess the ability to restrict the communication among processes such that communication is carried out in only one direction by reading only from specific processes such that no circular dependencies arise. Thereby, a strict hierarchy among the processes can be established by an algorithm. This potential of algorithms to establish a hierarchy among processes is important. It is later exploited to reason about the system decomposition.

2.2 Probabilistic influence

After having specified the deterministic system dynamics with processes, communication channels and algorithms, probabilistic influence is required.

The notions of determinism, probabilism and non-determinism

Two options to account for events as not being deterministic are *probabilistic* and *non-deterministic*. An event with one certain outcome is deterministic. An event is probabilistic or non-deterministic when it has more than one probable or possible outcomes. In the context of this thesis, an event like a coin flip or dice roll is

- deterministic when its outcome is certain,

- probabilistic when the probability for each *probable* outcome is known and

- non-deterministic when each *possible* outcome is known, but not the probability with which it occurs.

For instance, a coin toss event is deterministic when its outcome is known, for instance when the coin has two sides both showing heads. The event is probabilistic with multiple outcomes when the probabilities for all outcomes like both heads and tails are known and are not 0. The event is non-deterministic in case all possible outcomes are known but not the probabilities with which they occur.

Probability notation, time and constant influence

Single probabilities for outcomes of events are labeled $pr(\text{outcome})$. For instance, the outcome of a coin toss event can be described with $pr(\text{heads}) = 0.5$. The probability distribution over all probable outcomes of an event is a probability distributions characterized by a probability mass function $\sum_{\text{outcomes}} Pr(\{\text{outcomes}\}) = 1$. For instance, the distribution over the outcomes of a coin toss event is $Pr(\{\text{heads}, \text{tails}\}) = \{pr(\text{heads}), pr(\text{tails})\} = \{0.5, 0.5\}$. Index t labels events and probabilities with a timestamp, for instance $pr(\text{outcome}_t)$. We assume a discrete time model. All commands are atomic and their execution consumes one time step. The two probabilistic influences regarded in this thesis are scheduling and faults. Probabilistic scheduling is considered as second probabilistic influence to demonstrate that further probabilistic influence despite the fault model can be accounted for. The fault model is introduced next before the scheduling.

2.2.1   Fault model

A fault model specifies possible undesired influence perturbing the system such that it does not work according to its specification. This thesis focuses on *transient probabilistic* faults corrupting the register of an executing process with a given probability. These faults are also referred to as *sporadic* faults.

The thesis focuses on systems that are supposed to run indefinitely. When such a system is perturbed by a transient fault, it is designed to recover, that is, to converge to a desired behavior after the influence by transient faults stopped. In this context, certain faults like permanent and intermittent faults are not in the focus of this thesis. On the contrary, the goal is to evaluate how transient faults perturb a system in the long run, and how well the recovery of a system is able to cope with the effects of transient faults.

The focus on system properties *in the long run* further motivate to consider constant fault and scheduling probabilities. Systems that are designed for a limited mission time commonly exhibit a burn-in and burn-out phase, commonly known as *bathtub-curve*. The systems in the scope of this thesis are designed to run indefinitely and to be accessed at some arbitrary random time point. Since that time point is undetermined, it is not reasonable to consider probabilistic influence to be time-dependent.

The fault probability, which is the *general* probability that a sporadic fault occurs, is labeled $q = 1 - p$. The probability that a process executes without being perturbed by a fault is hence p. A fault causes the register of the *executing* process to store an arbitrary value within its register domain. For instance, a traffic light would not be able to store the value blue. The assumption that only the register of the executing process is prone to faults while the other processes are immune is motivated in the next paragraph "Immunity of algorithm and scheduler to faults".

In event of a fault q, the process stores and arbitrary value from the executing process domain. Each such probable outcome of that fault event is labeled $q = \{q_1, \ldots\}$. The set of fault outcomes is finite if the corresponding register domain is discrete and finite. The probability distribution over all probable fault free and faulty outcomes is labeled

$$Pr(Q) = \{pr(p), pr(q_1), pr(q_2), \ldots\} \tag{2.2}$$

such that the fault probability q is distributed among all possible fault outcomes $\sum_{i=1}^{|Q \backslash p|} pr(q_i)$ $= q$, with $|Q|$ being the cardinality of Q, the number of possible outcomes. We consider that faults perturb only the executing process' register and nothing else. This means that registers of non-executing processes are temporarily immune and algorithm as well as scheduler are immune, too.

Immunity of algorithm and scheduler to faults

The assumption that registers are susceptible to sporadic faults while the algorithms' susceptibility is negligible is realistic. For instance, embedded systems have *the algorithm* commonly burned to a mask ROM. A mask ROM cannot be changed by ordinary means once it is written. Contrary to the mask ROM, volatile data like intermediate results is commonly written into SRAM. Compared to SRAM, the liability to faults of a mask ROM is negligible.

Regarding the realism of employed models as discussed in the introduction, two recent studies by Schroeder et al. [Schroeder and Gibson, 2007, Schroeder et al., 2009] are noteworthy. The studies determine the fault susceptibility of volatile memory of real world systems, revealing that transient faults occur less frequently than originally anticipated. Assuming that also the scheduler is immune to faults is motivated by the focus on the fault tolerance of the system topology and the algorithm executed, whereas the degree to which the system's fault tolerance depends on the scheduling is not considered here.

Malign and benign faults

The outcome of a fault event q_i can be of either *malign* or *benign* nature. Malign here means that an illegal value is stored such that the system state violates safety conditions. Benign on the other hand means that a fault stores a legal value by chance. Notably, even if a fault causes to store a value that is different to what the algorithm would have stored, the result can still be benign, if it does not violate safety constraints, although it might *reroute* the execution trace in an unintended direction.

Fault, error, failure

Classifying the effects of undesired sporadic perturbations into faults, errors and failures is widely accepted and for instance classified by Avižienis et al. [Avižienis et al., 2004]. Yet, there are some controversies (e.g. [Denning, 1976][1]) about these definitions. This paragraph introduces a set of definitions tailored to suit the scope of this thesis that is based on the definitions by Avižienis et al. [Avižienis et al., 2004].

Definition 2.4 (Transient fault).
A transient fault temporarily (i.e. for finitely many computation steps) perturbs a process by manipulating its communication or computation, forcing it to store any arbitrary value.

The system recovers to a legal state when the fault is benign or when it is compensated before it is detected. For instance, if a process writes a faulty value into its register and overwrites it with a correct value before the faulty value effects the safety conditions. For instance, in the case of *write-after-faulty-write*, the fault is not read and thus it can be considered as undetected.

Definition 2.5 (Error).
An error is the possible consequence of a fault. A fault becomes an error when it is detected.

A system might be able to temporarily deprive its services from system user while errors are detected. A system recovers from errors when there are no more errors detected. Notably, Avižienis et al. [Avižienis et al., 2004] distinguish between "latent" undetected errors and detected errors. In the scope of this thesis, this distinction is not required. Errors are detected faults that are — on the pathway to becoming a failure — still tolerable.

---

[1]In this light, Denning [Denning, 1976] argues that the term "fault tolerance" is misleading and actually should be replaced by "error tolerance". Although his arguments are conclusive, the term *fault tolerance* has been coined over the past decades.

Definition 2.6 (Failure).
A failure is the possible consequence of an error. When an error is not compensated for in time, it becomes a failure.

On the escalation from correct operation to catastrophic failure, an error must violate desired constraints in order to be detectable. Yet, the possibility to successfully recover to an operable status *in time* still remains. For *some time*, errors can be tolerable in the sense that *there is still hope* that the system will recover and that the system will be able to provide the desired service with an acceptable delay.

Threat cycle

Figure 2.1 concludes this section by introducing the *threat cycle*:
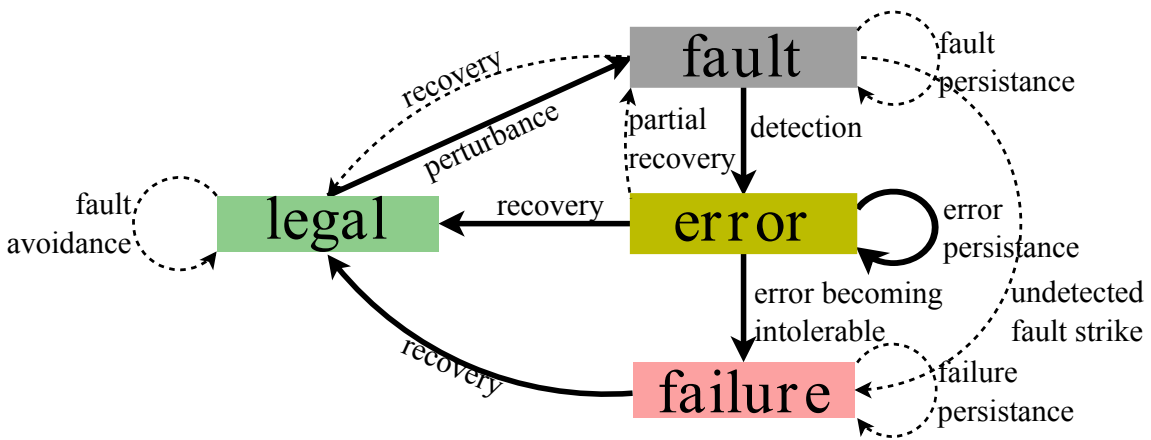
Figure 2.1: Threat cycle

The traffic light colors indicate the severity. The legal state is green, symbolizing that the system is up and running as expected. The fault stage is gray as it stands for those faulty states that the system cannot detect. The yellow state summarizes detected errors against which the system can deploy counter measures to recover. The red state marks failures.

The bold black arrows mark the transitions that are important in the context of this thesis. The transition from legal to fault states reflect the continuous probabilistic influence by the fault model. When a fault is detected, the system can actively try to compensate its effects. Until then, the error persists. In case the maximal admissible time span[2] for recovery is finite, the error becomes intolerable when that time runs out. In that case, the error becomes a failure. The system user can then be provided with a failure message and/or an incorrect result, depending on the system design.

Probabilistic faults

The examples in this thesis consider a probabilistic model for transient faults with a constant fault probability $q = 1 - p$. When a process π_i executes a computation step and is not perturbed by a fault, it deterministically executes one guarded command as specified

---

[2]Here, the *maximal admissible time span* means the amount of time that the user is willing to wait. That amount of time is *admitted* by the system user for recovery.

by its algorithm $\mathfrak{A}$. Otherwise, it stores a random value in its register. With c being the cardinality of the register domain — for instance $c = 3$ for a traffic light with three colors green, yellow and red —, one of these values is selected, each with an equal probability of $\frac{1}{c}$. This assumption can be arbitrarily adapted as desired.

2.2.2 Execution semantics and scheduling

While the — here deterministic — algorithm defines *what* the processes execute, scheduling and execution semantics determine *how* the processes execute.

Execution semantics

The term execution semantics, as for instance presented by Theel [Theel, 2000], specifies the cardinality of concurrent execution and its limitations. Processes executing concurrently or in parallel[3] introduce *parallel execution semantics* to the system, whereas processes executing one at a time introduce *serial execution semantics* to the system. The case when not every enabled process is allowed to execute, but more than one process is allowed to execute is referred to as semi-parallel. When every process with an enabled guard is continuously allowed to execute, the system executes under *maximal parallel execution semantics* as described for instance by Sarkar [Sarkar, 1993].

The examples in this thesis employ only serial execution semantics except for the case studies in chapter 7. It might initially seem that maximal parallel execution semantics are more general than serial execution semantics. On second thought, the processes share the common resource of execution right. Thereby, processes depend on each other. Serial execution semantics regards this issue while maximal parallel execution semantics does not. Thus, serial execution semantics allows for a more general discussion than maximal parallel execution semantics. The general examples before chapter 7 focus on *serial* execution semantics for two reasons: First, to allow for a more general approach than maximal parallel execution semantics, and second, to keep this approach comprehensive. The case studies in chapter 7 later explain the developed methods and concepts in the light of different kinds of execution semantics to further discuss this issue.

As discussed above, the algorithms of the examples are deterministic, meaning that always every process has exactly one guard enabled, and the commands are atomic, requiring exactly one discrete time step each. Combined with serial execution semantics, in each time step one process executes exactly one atomic command.

Scheduler

Distributed systems operate under schedulers controlling the execution sequence among the processes. With serial execution semantics and discrete time, the scheduler selects one process at each time step. The examples in this thesis are designed such that every process has exactly one guard enabled at each time step. In that context, the algorithms featured in this thesis are deterministic. Regarding the schedulers, the question arises, how the processes are selected to execute. The scheduler selections can be deterministic, probabilistic or non-deterministic. But what distinguishes a deterministic and a probabilistic scheduler?

---

[3] Armstrong [Armstrong, 2007] for instance distinguishes parallel execution as being synchronized from concurrent execution as being not synchronized. With a central scheduler demon and atomic execution steps considered in all examples, this thesis focuses on parallel execution.

Consider the goal of verifying that a system can recover from the effects of faults. Further consider a scheduler *probabilistically* putting all processes in a fixed sequence with each process occurring once, and *deterministically* calling the processes according to that order over and over again. Is the scheduler deterministic or probabilistic with regards to the goal of verifying the property of recovery?

One of the challenges to prove a system's fault tolerance is to show that certain required events eventually occur. The above scheduler selects every process within finite time, that is, within the length of the sequence each process is selected at least once. Since recovery commonly requires every process to execute finitely many times, the scheduler is suitable to prove that the system can recover after finitely many computation steps. Although the scheduler *randomly* selects the initial finite order, it *deterministically* selects the processes to execute once within each sequence. Regarding the verification of guaranteed finite recovery, the scheduler is considered to be deterministic. It allows to show that *every* execution trace of a certain length satisfies the desired property.

On the contrary, consider now a scheduler selecting a process to execute based on a *probabilistic* dice with the dice have as many sides as there are processes. The goal of every process executing eventually then cannot be verified as the scheduler can constantly ignore a process. The *probability* that a process is constantly ignored yet converges to zero over time and the probability for every process to be selected *some time* — without a specific upper temporal boundary — is 1. Regarding the verification of guaranteed finite recovery, this second scheduler is considered to be probabilistic. It does not allow to show that *every* execution trace of a certain length satisfies the desired property, but it allows to show that the accumulated probability of all infinite execution traces to satisfy the desired property is 1.

The schedulers considered in the examples are probabilistic in this context. They select the processes according to a uniform probability distribution that might as well be replaced by any other distribution. We label the event of scheduler selection with $\mathfrak{s}$, its outcome with $\mathfrak{s} = \pi_i$ such that for uniformly distributed scheduling $\forall \pi_i \in \Pi : pr(\mathfrak{s} = \pi_i) = \frac{1}{|\Pi|}$. We abbreviate $pr(\mathfrak{s} = \pi_i)$ with $\mathfrak{s}_i$.

Probabilistic faults, convergence and scheduling

This thesis considers a probabilistic fault model. With such a fault model, recovery cannot be shown for *every* execution trace as there are execution traces that continuously suffer from malign faults. When recovery cannot be shown for *every* execution trace anyway, one might consider a probabilistic scheduler for which deterministic recovery cannot be shown as well instead of a deterministic one, thereby accounting for a more general class of schedulers.

In this light, the scheduler is not part of the deterministic system model as per definition 2.1. It is modeled along with the fault model as *probabilistic environmental influence*. The goal of this thesis is yet to determine the tolerance of the *system* and not to consider the susceptibility of the environment to faults. In that context, the fault model is excluded from effecting the scheduler. Otherwise, the computed measure would not account for the fault tolerance of the *system*, but for the fault tolerance of the probabilistic scheduler as well.

2.3 Execution traces

Let $s_{i,t}$ be the state visited at time t and $\mathfrak{s}_t$ be the process selected by the scheduler at that time. Execution traces are sequences of states $\langle s_{i,t} \rightarrow s_{j,t+1} \ldots \rangle$ that the system traverses over time, as similarly specified in [Alpern and Schneider, 1985, Ebnenasir, 2005] and [Lynch, 1996, p.206]. A distinct infinite execution trace is referred to as σ^i and a distinct partial (or finite) execution trace within the interval $[t, t + k]$ is referred to as $\sigma^i_{t,k}$. Since different probabilistic events cause the system to visit different states, we include the events causing a transition between two subsequently visited states, annotated by the responsible event outcomes being the labels of the transition arrows: $\sigma^i = \langle s_{i,0} \xrightarrow{\mathfrak{s}_0 = \pi_i, p} s_{j,1} \xrightarrow{\mathfrak{s}_1 = \pi_j, q_3} \ldots \rangle$. Thereby, different execution traces traversing the same states can be distinguished, like for instance two traces where i) correct execution and ii) a benign fault have the same effect on the system. The transition from s_i to state s_j is abbreviated with $(\overrightarrow{s_i, s_j})$.

Each execution trace is the concatenation of outcomes of events, that is, probabilistic scheduler decisions and faults. With multiple outcomes being probable for each state at each time, the execution traces unfold like a tree structure over time. Execution traces distribute the probability mass of the system being in a certain state over all finitely many states of the state space as the execution progresses. With each computation step, the number of probable execution traces increases. In the limit[4], there are uncountably infinitely many[5] execution traces. Each of them is *improbable*, meaning they all have zero probability. Their accumulated probability is 1. The following coin-toss example explains this.

Assume the simple system of one process storing heads when it is not perturbed by a fault and tails otherwise. Consider that the set of initial states contains both states $\mathcal{S}_0 = \{\langle \text{heads} \rangle, \langle \text{tails} \rangle\}$. The two shortest execution traces both contain one of the states. After one time step, the execution traces $\sigma^i_{0,1} \in \{\langle \text{heads} \xrightarrow{p} \text{heads} \rangle, \langle \text{heads} \xrightarrow{q} \text{tails} \rangle, \langle \text{tails} \xrightarrow{p} \text{heads} \rangle, \langle \text{tails} \xrightarrow{q} \text{tails} \rangle\}$ are probable, their specific probabilities can be computed. After n time steps and $|\mathcal{S}| = 2$, there are 2^n probable execution traces. After infinitely many time steps, which means after concatenating infinitely many outcomes, each of the uncountably infinitely many execution traces is *improbable* but *possible*.

Abbreviations for outcome probabilities

We further introduce the following notational abbreviations. Consider the probability for all specific faults q_i to be uniformly distributed. The probability that a process is selected to execute and is corrupted by a specific fault q_j is then $\mathbf{q} = \mathfrak{s}_i \cdot pr(q_j)$ for all faults, and the probability it executes correctly is $\mathbf{p} = \mathfrak{s}_i \cdot p$. With uniformly distributed scheduling and fault probabilities, $\mathbf{p}$ and $\mathbf{q}$ are equal for every process.

Reachability of states

The state space comprises all reachable states. To show that every state is reachable from every other state within finitely many steps with a non-zero probability, the fault

---

[4]The *limit* refers to infinitely many preceding execution steps that have been executed.

[5]The execution traces are a bijection to the real numbers. Thereby, they are uncountable. For instance, assume a one process system in which the process randomly stores one digit from 0 to 9, initially storing 0. That way, each positive real number between 0 and 1 can be generated. this holds for every probabilistic system with infinite execution traces and more than one state.

model suffices. Every time step, one process is randomly selected and *probably* stores any value with a certain probability. Every finite sequence of scheduler selections is probable, including all those of the same length as there are processes in the system. In all these sequences, there are sequences in which every process is selected exactly once. In each such sequence, every process probably changes the value stored in its register to any arbitrary value. Thus, every state is probably reachable from every other state within finitely many computation steps.

After introducing deterministic system dynamics and probabilistic environmental models, the discussion about execution traces and reachability allows to discuss the construction of a transition model.

2.4   From system model to transition model

The state space $\mathcal{S}$ contains the states that the system can reach. A finite amount of outcomes of probabilistic events is responsible for the transition between each two states s_i and s_j, thus determining the transition probability $pr(\overrightarrow{s_i, s_j})$, which is the probability to jump from s_i to s_j within one computation step, with $pr(\overrightarrow{s_i, s_j}) : \mathcal{S} \times \mathcal{S} \mapsto [0, 1]$. A transition between two states s_i and s_j is probable when there exists at least one event with a corresponding outcome that is responsible for that transition. For each state $s_i \in \mathcal{S}$, the outgoing transition probabilities accumulate to one: $\forall s_i, s_j \in \mathcal{S} : \sum_{s_j} pr(\overrightarrow{s_i, s_j}) = 1$. Each transition probability can be computed with the accumulated[6] scheduling and fault probabilities while accounting for the algorithm and the topology. Consider a quadratic transition matrix $\mathcal{M}$ with $|\mathcal{S}|$ rows and columns, such that the element in the i-th row and the j-th column is the transition probability between the corresponding states $\mathcal{M}_{i,j} = pr(\overrightarrow{s_i, s_j})$.

Notably, we consider probabilistic influence like fault and scheduling probabilities to be constant. If it was time-dependent, the transition matrix would have to be computed for each time step. A relaxation to this assumption is discussed in chapter 8. Furthermore, under serial execution semantics, only transitions between states that differ in not more than one register are probable. For maximal parallel execution semantics on the other hand, every state is reachable from any other state within one computation step. Serial execution semantics lead to a sparse matrix while maximal parallel execution semantics result in a dense matrix. The same holds for fault models which allow the corruption of non-executing processes.

Due to discrete computation steps and serial execution semantics, a discrete time transition model is selected. Furthermore, the register domains are considered to be finite and discrete, meaning that the state space is finite. Thus, *discrete time Markov chains* as introduced for instance in [Kemeny and Snell, 1976, Norris, 1998] and [Baier and Katoen, 2008, p.747] are selected as transition model.

Definition 2.7 (Discrete time Markov chain [Müllner et al., 2013])**.**
A discrete time Markov chain is a tuple $\mathcal{D} = \{\mathcal{S}, \mathcal{M}, Pr_0(\mathcal{S})\}$ where

- *$\mathcal{S}$ is a countable, nonempty set of states,*

- *$\mathcal{M} = Pr(\mathcal{S} \times \mathcal{S}), Pr : \mathcal{S} \times \mathcal{S} \mapsto [0, 1]$ is the transition probability matrix*

---

[6]Since multiple events — like benign faults and correct execution — can trigger the same transitions, the transition probabilities accumulate the respective probabilities of all those events responsible for them.

- *and $Pr_0(\mathcal{S})$, $Pr : \mathcal{S} \mapsto [0, 1]$ is the initial probability distribution at time $t = 0$.*

The DTMCs constructed in the context of this thesis have finite state spaces. The number of states of $\mathcal{D}$, which is the cardinality of the DTMC, is denoted by $|\mathcal{S}|$. The vertices of $\mathcal{D}$ are the states in $\mathcal{S}$. The probability mass in state s_i at time t is denoted as $pr_t(s_i) \mapsto [0, 1]$ and the probability distribution at time t with $Pr_t(\mathcal{S}) = \{pr_t(s_1), \ldots\}$.

Ergodicity of the DTMC

Suitable introductions to Markov chain theory are provided by Kemeny and Snell [Kemeny and Snell, 1976], Norris [Norris, 1998] and Baier and Katoen [Baier and Katoen, 2008]. A state s_i in $\mathcal{S}$ is ergodic if it is aperiodic and positive recurrent. A DTMC $\mathcal{D}$ is ergodic when it is irreducible and only contains ergodic states. This means that a Markov chain is ergodic if every state is reachable from any other state.

With probabilistic scheduling and transient faults that can cause an executing process to store any arbitrary value, each state is reachable within finitely many steps from every other state. Thereby, the DTMC resolving from the deterministic system dynamics and the model of transient faults and the probabilistic scheduler is ergodic.

Why is this important? The following chapter motivates to focus on recovering systems that are designed to run indefinitely. This thesis assumes that the system is initially upon user request in a state with a probability according to the stationary distribution. With their DTMC being ergodic, their *limiting probability distribution* (or stationary distribution) over the state space converges to a specific distribution in the limit. To measure the probability with which the system satisfies desired constraints, we assume that the system user accesses the system at an arbitrary time point set to the limit. The methods and concepts account for any arbitrary initial configuration or probability distribution. Yet, the stationary distribution is the most reasonable assumption in the context of non-terminating recovering systems.

Hamming distance

In graph theory, the Hamming distance, as introduced by Hamming [Hamming, 1950] in 1950 and similarly also by Golay [Golay, 1949] in 1949, is the number of vertices on the shortest path between two vertices. Here, the vertices are states in a DTMC. An execution trace σ^i traverses the states of the (finite) state space $\mathcal{S}$ of a system according to the probabilities that are specified by its transition model. Under *serial execution semantics*, at most one process changes its register per time step. Therefore, two successive states $s_{i,t}, s_{j,t+1} \in \sigma^i$ can differ in at most one register. Only such transitions between states are probable that differ in at most one register.

In this context, we refer to the *number of registers* that can at most change per execution step as *Hamming distance*. Consider the simple traffic light example with a transition model as shown in figure 2.2.
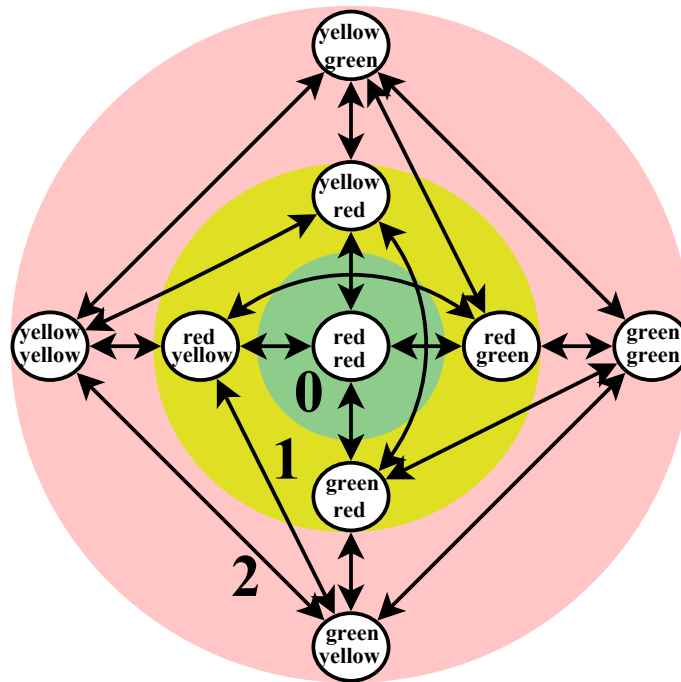
Figure 2.2: Simple traffic lights transition model demonstrating Hamming distance

Assume that the current state of two traffic lights is ⟨red,red⟩. Within Hamming distance 1, demarcated as yellow area, lie the states ⟨red,red⟩ since distance 0, demarcated as green area, lies within distance 1, ⟨yellow,red⟩, ⟨red,green⟩, ⟨green,red⟩, and ⟨red,yellow⟩. This means that at least one register must remain red. Increasing the Hamming distance of a system means to allow the system to reach a broader spectrum of states within one time step. For instance, allowing transitions between states differing in two registers means a Hamming distance of two. The maximal Hamming distance in that sense coincides with the number of registers that can possibly change. Values greater than the maximal Hamming distance are futile.

2.5   Example - traffic lights

Before the system model is used to compute fault tolerance properties, a brief example demonstrates how a system can be modeled and how a transition model can be derived on a pedestrian crossing, based on an example by Baier and Katoen [Baier and Katoen, 2008, p.90] and shown in figure 2.3. It contains two intersecting paths, a road for cars and a pedestrian passage. Each path is controlled by two traffic lights, one for each direction, to exclude simultaneous access by both cars and pedestrians. Each pair of traffic lights handling the access for one passage is controlled by one process, either π_1 or π_2, and is accessing the same registers. For simplicity, all processes and traffic lights are assumed to be equal, that is, pedestrians also see a yellow light.

System model

Figure 2.3 shows the schematics of the crossing.
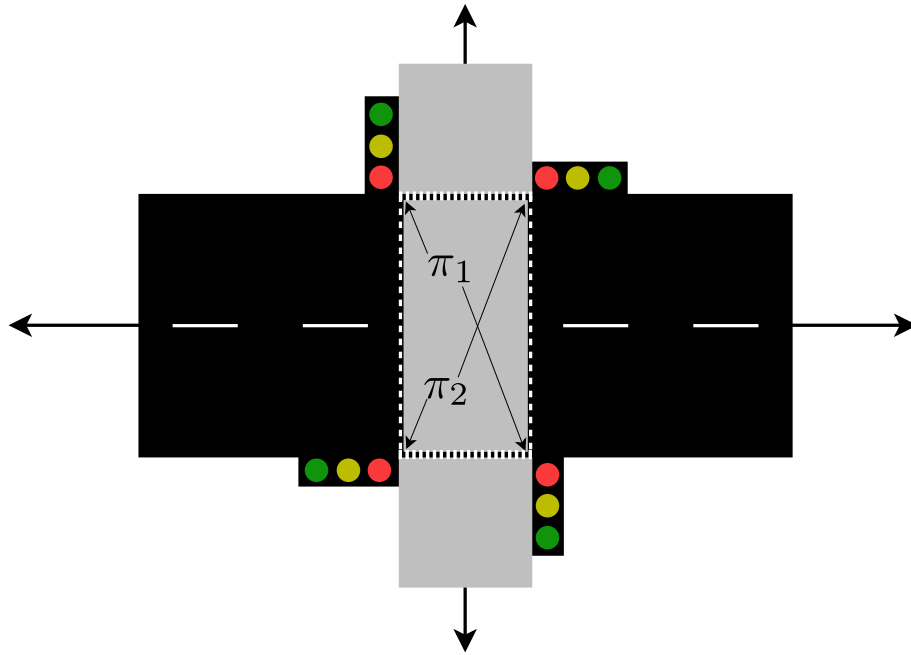
Figure 2.3: Pedestrian crossing

Pedestrians look at traffic lights of process π_1 and cars at traffic lights of process π_2. The traffic lights of each process are considered to be mutually consistent. Either the cars or the pedestrians are allowed to exclusively access the crossing. Situations in which both have simultaneous access are prohibited. A system $\mathfrak{S} = \{\Pi, E, \mathfrak{A}\}$ contains two processes $\Pi = \{\pi_1, \pi_2\}$ that are connected via one communication channel $E = \{e_{1,2}\}$ as shown in figure 2.4.

Figure 2.4: Topology of two processes in the traffic light example

The communication channel refers to the processes having mutual read access. The algorithm $\mathfrak{A}$ controlling both traffic lights is shown in algorithm 2.1. The scheduler selects at each time step one of the two processes at random, each with a probability of $\mathfrak{s}_i = 0.5$. The algorithm is referred to as *traffic lights algorithm* (TLA).

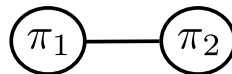| Label	Guard enabled and $\mathfrak{s} = \pi_1$		Command	Label	Guard enabled and $\mathfrak{s} = \pi_2$		Command
a_1	$R_1 = green$	$\wedge R_2 = green$	$R_1 := red_1$	a_{26}	$R_1 = green$	$\wedge R_2 = green$	$R_2 := red_1$
a_2	$R_1 = green$	$\wedge R_2 = yellow$	$R_1 := red_1$	a_{27}	$R_1 = green$	$\wedge R_2 = yellow$	$R_2 := red_1$
a_3	$R_1 = green$	$\wedge R_2 = yellow_1$	$R_1 := red_1$	a_{28}	$R_1 = green$	$\wedge R_2 = yellow_1$	$R_2 := red_1$
a_4	$R_1 = yellow$	$\wedge R_2 = green$	$R_1 := red_1$	a_{29}	$R_1 = yellow$	$\wedge R_2 = green$	$R_2 := red_1$
a_5	$R_1 = yellow_1$	$\wedge R_2 = green$	$R_1 := red_1$	a_{30}	$R_1 = yellow_1$	$\wedge R_2 = green$	$R_2 := red_1$
a_6	$R_1 = green$	$\wedge R_2 = red$	$R_1 := red$	a_{31}	$R_1 = green$	$\wedge R_2 = red$	$R_2 := red_1$
a_7	$R_1 = green$	$\wedge R_2 = red_1$	$R_1 := yellow_1$	a_{32}	$R_1 = green$	$\wedge R_2 = red_1$	$R_2 := red_1$
a_8	$R_1 = red$	$\wedge R_2 = green$	$R_1 := red_1$	a_{33}	$R_1 = red$	$\wedge R_2 = green$	$R_2 := red$
a_9	$R_1 = red_1$	$\wedge R_2 = green$	$R_1 := red_1$	a_{34}	$R_1 = red_1$	$\wedge R_2 = green$	$R_2 := yellow_1$
a_{10}	$R_1 = yellow$	$\wedge R_2 = yellow$	$R_1 := red_1$	a_{35}	$R_1 = yellow$	$\wedge R_2 = yellow$	$R_2 := red_1$
a_{11}	$R_1 = yellow$	$\wedge R_2 = yellow_1$	$R_1 := red_1$	a_{36}	$R_1 = yellow$	$\wedge R_2 = yellow_1$	$R_2 := red_1$
a_{12}	$R_1 = yellow_1$	$\wedge R_2 = yellow$	$R_1 := red_1$	a_{37}	$R_1 = yellow_1$	$\wedge R_2 = yellow$	$R_2 := red_1$
a_{13}	$R_1 = yellow_1$	$\wedge R_2 = yellow_1$	$R_1 := red_1$	a_{38}	$R_1 = yellow_1$	$\wedge R_2 = yellow_1$	$R_2 := red_1$
a_{14}	$R_1 = yellow$	$\wedge R_2 = red$	$R_1 := red$	a_{39}	$R_1 = yellow$	$\wedge R_2 = red$	$R_2 := red_1$
a_{15}	$R_1 = yellow_1$	$\wedge R_2 = red$	$R_1 := red$	a_{40}	$R_1 = yellow_1$	$\wedge R_2 = red$	$R_2 := red_1$
a_{16}	$R_1 = yellow$	$\wedge R_2 = red_1$	$R_1 := green$	a_{41}	$R_1 = yellow$	$\wedge R_2 = red_1$	$R_2 := red_1$
a_{17}	$R_1 = yellow_1$	$\wedge R_2 = red_1$	$R_1 := red$	a_{42}	$R_1 = yellow_1$	$\wedge R_2 = red_1$	$R_2 := red_1$
a_{18}	$R_1 = red$	$\wedge R_2 = yellow$	$R_1 := red_1$	a_{43}	$R_1 = red$	$\wedge R_2 = yellow$	$R_2 := red$
a_{19}	$R_1 = red$	$\wedge R_2 = yellow_1$	$R_1 := red_1$	a_{44}	$R_1 = red$	$\wedge R_2 = yellow_1$	$R_2 := red$
a_{20}	$R_1 = red_1$	$\wedge R_2 = yellow$	$R_1 := red_1$	a_{45}	$R_1 = red_1$	$\wedge R_2 = yellow$	$R_2 := green$
a_{21}	$R_1 = red_1$	$\wedge R_2 = yellow_1$	$R_1 := red_1$	a_{46}	$R_1 = red_1$	$\wedge R_2 = yellow_1$	$R_2 := red_1$
a_{22}	$R_1 = red$	$\wedge R_2 = red$	$R_1 := red_1$	a_{47}	$R_1 = red$	$\wedge R_2 = red$	$R_2 := red$
a_{23}	$R_1 = red_1$	$\wedge R_2 = red$	$R_1 := red$	a_{48}	$R_1 = red_1$	$\wedge R_2 = red$	$R_2 := yellow$
a_{24}	$R_1 = red$	$\wedge R_2 = red_1$	$R_1 := green$	a_{49}	$R_1 = red$	$\wedge R_2 = red_1$	$R_2 := red$
a_{25}	$R_1 = red_1$	$\wedge R_2 = red_1$	$R_1 := yellow$	a_{50}	$R_1 = red_1$	$\wedge R_2 = red_1$	$R_2 := green$

Algorithm 2.1: The traffic lights algorithm (TLA)

The algorithm does not comply with the three aspect standard sequence $\langle red \rangle \rightarrow \langle red$ and $yellow \rangle \rightarrow \langle green \rangle$ is replaced by $\langle red \rightarrow yellow \rightarrow green \rangle$ in the algorithm.

Although a set of common traffic lights shows only three colors, this example requires five colors. Consider the system to be in a state where both processes store red. Since the scheduler can be probabilistic, it is undetermined which process is next to execute. Hence, a second red value is required to indicate which process is next to proceed to green, independent of which choice the scheduler makes. Furthermore, a second yellow light is required. In the constellation of one light showing red and the other showing yellow, the process showing yellow would not know if next to proceed to green or red. This extension is required since Markov chains have no memory and *fairness*[7] among the road users is required. The extra values are explained in detail in paragraph "Probabilistic scheduling and algorithmic sequencing" on page 20.

The algorithm executes the following loop sequence over and over again:

$$\ldots \rightarrow \langle red_1, red \rangle \rightarrow \langle red_1, yellow \rangle \rightarrow \langle red_1, green \rangle \rightarrow \langle red_1, yellow_1 \rangle \rightarrow$$
$$\langle red_1, red_1 \rangle \rightarrow \langle yellow, red_1 \rangle \rightarrow \langle green, red_1 \rangle \rightarrow \langle yellow_1, red_1 \rangle \rightarrow \qquad (2.3)$$
$$\langle red, red_1 \rangle \rightarrow \langle red, red \rangle \rightarrow \langle red_1, red \rangle \rightarrow \ldots$$

When the system is in a state that does not belong to that sequence and executes a fault free step, it immediately reaches a state of the sequence as shown in figure 2.5. The system *converges* to that sequence.

---

[7]Fairness here refers to the alternating access to the intersection, cf. section 3.1.2.

The loop sequence described in equation 2.3 models the behavior one would expect[8] from a set of traffic lights. The corresponding transition model is shown in figure 2.5. The colors are abbreviated with their initial letter. The bold black arrows show the loop of equation 2.3. The states in that loop are colored: the left part shows the value of R_1 and the right part shows the value[9] of R_2. Furthermore, the blue light arrows show the convergence towards the states of the loop. Self-targeting transitions are not shown for a better visibility.
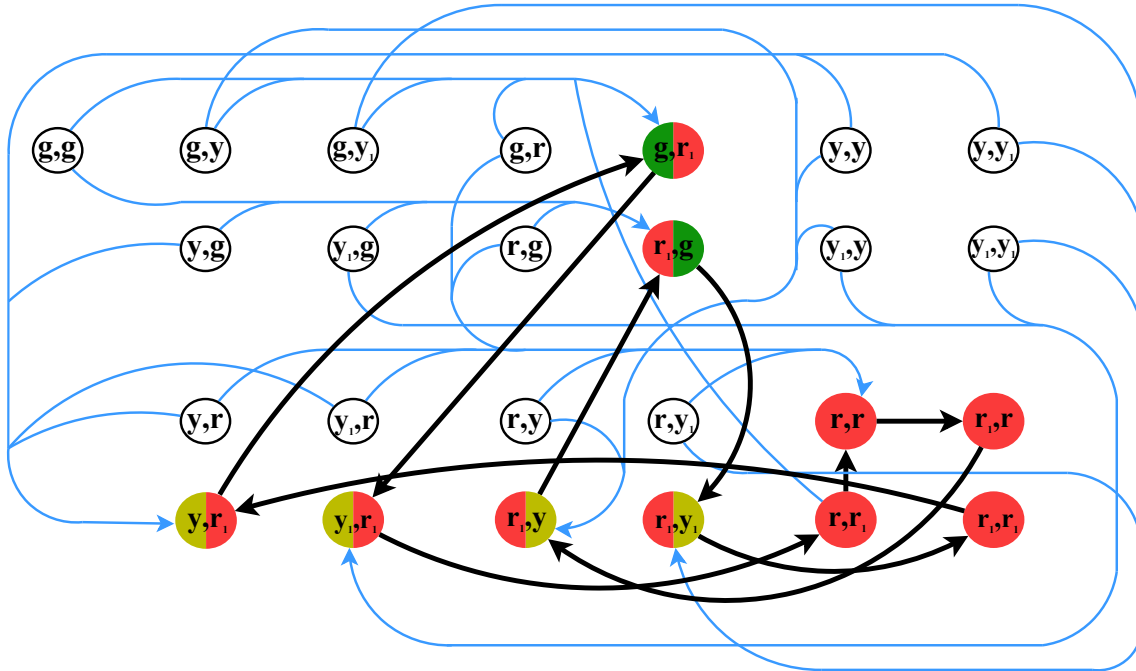
Figure 2.5: Algorithm transitions

The system does not reach a state outside the loop sequence deliberately. When the system is initially in a state within the loop, the guarded commands controlling the loop from formula 2.3 — including the self-targeting transitions — guarantee *algorithmic closure* during the absence of faults. Algorithmic closure means that the algorithm is closed with regards to a specific set of states that in this example form a loop. It cannot reach a state not belonging to that set solely with fault free execution $\mathfrak{A}$.

In case the system is not *in the loop*, execution of any process lets the system reach a state within the loop within one computation step, providing *convergence*[10]. Convergence is the property of a system to reach a set of states — here the states of the loop — within finitely many computation steps. This property can here be guaranteed even with probabilistic schedulers since the algorithmic Hamming distance from *every* state outside the loop sequence to a state within the loop sequence is 1.

---

[8]To be precise, a probabilistic scheduler might continuously select the same process over and over again, thus leaving the system in the same state. Other than that, this system is as close as possible to a real traffic light considering a probabilistic scheduler.

[9]Different yellow and red tones are not distinguished.

[10]This special case coincides with the notion of *snap stabilization* [Delporte-Gallet et al., 2007, Tixeuil, 2009]

To show convergence for a system operating under a probabilistic scheduler is a special property that includes i) probabilistic scheduling, ii) only two processes being involved, and iii) the leverage that only one *arbitrary* process is required to execute one fault free computation step to converge to the loop.

This concludes the fault tolerance aspects of the TLA example. The next paragraph discusses the functional issue that both parties should get alternating access to the crossing.

Proving interleaving access in spite of probabilistic scheduling

A probabilistic *coin-flip* scheduler randomly selects one of the two processes each with a probability of $\mathfrak{s}_1 = \mathfrak{s}_2 = 0.5$. As discussed on page 7, the algorithm has the ability to enforce hierarchy and order among the processes and their execution. Here, the algorithm exploits two additional values — one yellow and one red — to establish an alternating access, also referred to as *interleaving* access, of both cars and pedestrians, provided the system is in a state within the loop sequence. The proof that access is interleaving among the parties is discussed informally. Assume that the system is in any arbitrary state outside the loop sequence. Then, the system reaches a state within the loop sequence within one computation step, regardless which process is selected by the scheduler. After the system converged to a state within the loop sequence, interleaving access is desired. In absence of faults, a process can only change its register when the other process did execute since its own last execution. Otherwise, it will only store the value its register already stores according to algorithm 2.1. If the particular other process executed directly before a process executes, the system must be in a state for which the registers enable a guarded command that will change the value stored in the executing process' register. Thereby, the algorithm guarantees alternating access between the parties.

Notably, — for the same reason the system guarantees alternating access in spite of probabilistic scheduling — it also provides the same functionality under both serial and parallel execution semantics, considering that the registers are read at the beginning of each computation step and written at its end.

Fault model

Next, the probabilistic fault model is specified. We select the probability for a transient fault in this example to be $q = 0.25$ and $p = 0.75$ as numerical values. In average, every fourth execution is perturbed by a transient fault. Any other probability distribution works as well. In this case, traffic lights are made of unreliable components and operate in a very hostile environment. When perturbed by a fault, the executing process writes a random value of its domain to its register. We assume that faults can be of either malign or of benign nature. The domain of each process' register comprises five values: *green*, *yellow*, *yellow*$_1$, *red* and *red*$_1$. A fault causes a process to store one of these values at random, each with an equal probability. The probability that a specific process is selected and executes without corruption is

$$\mathbf{p} = \mathfrak{s}_i \cdot p = 0.5 \cdot 0.75 = 0.375 \tag{2.4}$$

The probability for a specific process to be selected and to execute with a corruption is

$$1 - (|\Pi| \cdot \mathbf{p}) = 0.25 \tag{2.5}$$

The probability that a specific process is selected and executes with a specific corruption is

$$\mathbf{q} = \mathfrak{s}_i \cdot pr(q_i) = 0.5 \cdot 0.25 \cdot 0.2 = 0.025 \tag{2.6}$$

Since both scheduling and fault distributions are uniformly distributed among the processes and faults, they are not particularly considered in the variables.

DTMC

The transition probability matrix $\mathcal{M}(\mathcal{S} \times \mathcal{S})$ shown in table 2.1 contains the transition probabilities between each state pair for all 25 states. The colors are abbreviated with their initial letters, that is, g for green, y for yellow and r for red. For now, the symbolic DTMC suffices. The numerical DTMC in which the variables are replaced with their numerical values is required later and shown in table 4.1.

↓from/to→	g,g	g,y	g,y_1	y,g	y_1,g	g,r	g,r_1	r,g	r_1,g	y,y	y,y_1	y_1,y	y_1,y_1
g,g	2q	q	q	q	q	q	p+q	q	p+q				
g,y	q	2q	q			q	p+q			q		q	
g,y_1	q	q	2q			q	p+q				q		q
y,g	q			2q	q			q	p+q	q	q		
y_1,g	q			q	2q			q	p+q			q	q
g,r	q	q	q			2q	p+q						
g,r_1	q	q	q			q	p+2q						
r,g	q			q	q			2q	p+q				
r_1,g	q			q	q			q	p+2q				
y,y		q		q						2q	q	q	
y,y_1			q	q						q	2q		q
y_1,y		q			q					q		2q	q
y_1,y_1			q		q						q	q	2q
y,r				q		q				q	q		
y_1,r				q		q						q	q
y,r_1				q			p+q			q	q		
y_1,r_1				q			q					q	q
r,y		q						q		q		q	
r,y_1			q					q			q		q
r_1,y		q						p+q		q		q	
r_1,y_1			q						q		q		q
r,r						q		q					
r_1,r							q						
r,r_1						q		q					
r_1,r_1							q		p+q				

↓from/to→	y,r	y_1,r	y,r_1	y_1,r_1	r,y	r,y_1	r_1,y	r_1,y_1	r,r	r_1,r	r,r_1	r_1,r_1
g,y					q		p+q					
g,y_1						q		p+q				
y,g	q		p+q									
y_1,g		q		p+q								
g,r	q	q							p+q	q		
g,r_1			q	p+q							q	q
r,g					q	q			p+q	q	q	
r_1,g							q	p+q		q		q
y,y	q		p+q		q		p+q					
y,y_1	q		p+q			q		p+q				
y_1,y		q	p+q	q			p+q					
y_1,y_1		q	p+q	q			p+q					
y,r	2q	q	p+q	p+q					p+q	q		
y_1,r	q	2q		p+q					p+q	q		
y,r_1	q		p+2q	q							q	q
y_1,r_1		q	q	p+2q						p+q		q
r,y					2q	q	p+q		p+q	q		
r,y_1					q	2q		p+q	p+q	q		
r_1,y					q		p+2q	p		q		q
r_1,y_1						q	q	p+2q				p+q
r,r	q	q			q	q			p+2q	p+q	q	
r_1,r	q	q					p+q	q	q	p+2q		q
r,r_1			q	q	q	q				p+q	p+2q	q
r_1,r_1			p+q	q			q	q		q	q	2q

Table 2.1: Symbolic transition matrix $\mathcal{M}$ of the TLA

The *left* half of $\mathcal{M}$ is shown in the upper part and the *right* half on the lower part, skipping the first empty row in the latter one. The transitions in the blue cells represent the loop sequence from formula 2.3, that is, the *algorithmic progress*. The green cells model the remaining transitions by the algorithm and benign faults, that is, the *recovery property* — also referred to as *convergence* — of the system to reach a state within the loop sequence. The red cells model malign faults with the dark red cells being twice as probable as the light red cells. The DTMC is obviously ergodic since every state is reachable from every other state.

2.6   Summarizing the system model

This chapter introduced the system topology and algorithm and distinguished deterministic system dynamics from probabilistic environmental influence. Execution traces were discussed and the construction of a transition model — discrete time Markov chains — was presented. A simple example demonstrated the modeling of a real world system within a probabilistic environment along with the construction of the corresponding transition model.

3. Fault tolerance terminology and taxonomy

This chapter introduces the necessary fault tolerance terminology and discusses how the relevant terms are related. Since the "Notes on Digital Coding" by Golay from 1949 [Golay, 1949], fault tolerance related terms have often been defined for specific purposes. Since then there has been an ongoing process to establish a general taxonomy [Becker et al., 2006, Rus et al., 2003]. This chapter provides one taxonomy, based on an article by Avižienis et al. [Avižienis et al., 2004], that is consistent with the scope of this thesis.

Setting

The setting we assume is simple. A user requests a service from an interactive system. The system is designed to run permanently. It provides an answer in response to the user request. The system runs in a hostile environment in which it is exposed to transient faults. The system has the ability to recover from the effects of such faults. Its response to the system user is considered as being correct when no effects of transients faults are present in the system. A safety predicate specifies if the system operates correctly or if effects of a fault are present. The system provides incorrect answers when the effects of faults are present, and correct answers otherwise. A detector can be mounted between user and system to check responses for their correctness according to the predicate. In case an incorrect response is detected, the system service can be deprived from the user until the system provides a correct answer again.
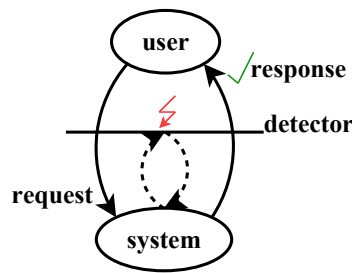
Figure 3.1: A user requesting system service

The system user is not willing to wait indefinitely for a correct answer. With system and transition models — as established in the previous chapter — at hand, the goal is to determine how well the system can provide a correct service in a hostile environment *in time*. To achieve this goal, a novel fault tolerance measure is proposed in the following chapter that suits the scope of this thesis, that is, measuring how well a system provides its service and how well it recovers.

Structure of this chapter

A taxonomy of terms that are required to discuss fault tolerance is introduced in section 3.1 and the individual terms are defined. The concept of self-stabilization and its variants that are required in the context of this thesis are discussed in section 3.2. Section 3.3 reflects on the *static* distinction of deterministic fault tolerance types to set them in the light of the *dynamic* aspects of probabilistic recovery in section 3.4. Section 3.5 exploits this view to introduce *unmasking fault tolerance* with the previously established terminology. Section 3.6 concludes the key aspects of this chapter.

3.1   Definitions

In this section, a *fault tolerance taxonomy* is proposed that suits the scope of this thesis. The terms it contains are defined. The example of the TLA is continued. The following taxonomy contains the terms that are important for this thesis[1]:

---

[1]Notably, fault tolerance can be perceived as part of dependability and be embedded into a broader context. Further examples of how relevant terms can be connected are shown in appendix A.4.1 on pages 157 ff.
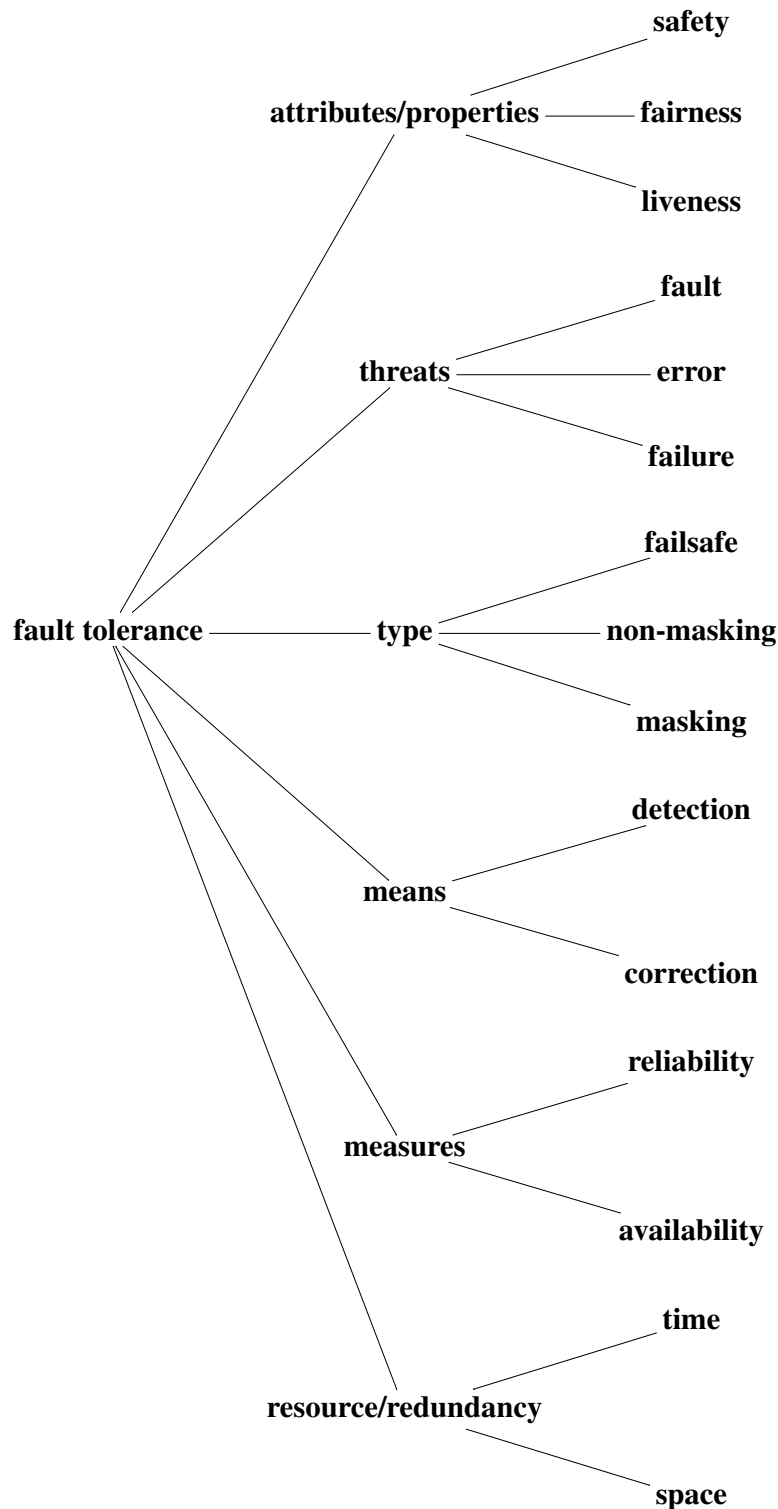
Figure 3.2: Fault tolerance taxonomy (not exhaustive)

The fault tolerance taxonomy shown in figure 3.2 is not exhaustive. Further *leafs*[2] and *branches* might be added to the tree. The goal of fault-tolerant systems is to provide for desired properties such as safety, fairness or liveness. Threats put these properties at risk. The fault tolerance type specifies whether safety or liveness are allowed to be (temporar-

---

[2]Due to the tree shape of the taxonomy, fault tolerance is referred to as *root*, the middle layer as *branches* and the terms on the right hand side are referred to as *leafs*.

ily) violated. Means provide functionalities to increase the chances of satisfying desired properties, for instance by lowering the risk that a fault succeeds with respect to the type. Measures allow to address how well a system manages to satisfy desired properties with regards to threats and type, and supported by means. The means to increase the fault tolerance of a system can be provided via spatial or temporal redundancy, which are also referred to as the *currencies* of fault tolerance to pay for certain means. The remainder of this section discusses the *leaf* terms from top to bottom.

3.1.1   Safety

Popular definitions of safety in literature are provided in appendix A.4.3 for comparison. Informally, safety means that "the bad thing" does not happen [Lamport, 1977]. We define safety in the context of the thesis as state predicate:

Definition 3.1 (State safety).
A system $\mathfrak{S}$ is in a safe state s_i with regards to a safety predicate $\mathcal{P}$ when that state satisfies the safety predicate.

This thesis refers to *safety* with its invariant based notion of *state safety* for brevity. A safety predicate is a Boolean expression over (a subset of) process registers. It partitions the state space into legal states satisfying $\mathcal{P}$ and illegal states violating $\mathcal{P}$. Formally, safety is expressed as $s_i \models \mathcal{P}$. A system violating the safety predicate $s_i \not\models \mathcal{P}$ can possibly reach a state from which it satisfies $\mathcal{P}$ at a later time[3].

Alternative definitions from literature consider safety for execution traces or to be final in the sense that once safety is violated the system cannot reach a legal state anymore. Both, an extension to cover for execution traces as well as the co-evaluation of mixed mode faults like permanent and transient, are discussed in the future work section in chapter 8.

3.1.2   Fairness

Definitions of fairness from literature are concluded in appendix A.4.4. Informally, fairness means that every process that can execute is selected to execute eventually, that is, within finite time or within a finite number of execution steps.

Definition 3.2 (Fairness).
Fairness means that every process that is enabled infinitely often is selected infinitely often by the scheduler.

Finite terminating sequences are necessarily fair as no enabled process is neglected forever [Manna and Pnueli, 1981a, p.246]. *Weak* fairness means that a process must be *continuously* enabled, that is, without possible interruptions, whereas *strong* fairness means that a process must be *continually* enabled, that is, with possible interruptions [Lamport, 2002].

From the system execution perspective this notion — strong and fair — seems unintuitive as the processes fulfill a *stronger* requirement being continuously available compared to their continually enabled counterpart. From the scheduling perspective yet, this notion is

---

[3]The perception that systems can recover from safety violations (cf. e.g. [Alpern and Schneider, 1985], quoted in A.4.3 on page 160) contradicts the perception that faults are *remediable* as advocated in this thesis.

right. The scheduler can be *weaker* as all processes are *continuously* enabled until they are selected. This distinction, that *weak* goes with *continuously* and *strong* with *continually* is important to the remainder of this thesis.

Every weakly fair sequence is also strongly fair, but not vice versa. Hence, the class of weakly fair systems is contained in the class of strongly fair systems. Similarly, the possible execution traces caused by the classes of i) deterministic schedulers and ii) probabilistic schedulers with a finite horizon are contained within the possible execution traces caused by probabilistic schedulers with an infinite horizon as discussed in section 2.2.2. The section also states that the examples in this thesis consider *continuously* enabled processes.
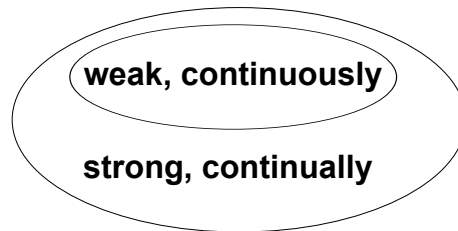
Figure 3.3: Weak fairness is a subset of strong fairness

With a probabilistic scheduler the fairness assumption is relaxed as follows:

Definition 3.3 (Probabilistic fairness)**.**
Probabilistic fairness means that every process that is enabled infinitely often from a time step onwards is selected with probability 1 by the scheduler.

The probabilistic relaxation is applicable to both weak and strong fairness: A process that is infinitely often continuously (strong) or continually (weak) enabled is selected with probability 1 by the scheduler. The difference between fairness and probabilistic fairness is vital and returns also for differentiating between self-stabilization and probabilistic self-stabilization in section 3.2. A probabilistic scheduler might continuously ignore one process. Although the probability for such a trace decreases over time and is 0 in the limit, such traces are *possible*, but not *probable*. Thus, referring to fairness *only* with probability 1 does not suffice to *verify* fairness since (at least) one possible counter example can be provided like the one in which one process is infinitely neglected by the scheduler.

The deterministic system dynamics of the examples in this thesis rely on weak fairness. But why must every process have deterministically one guard continuously enabled? When fairness can be discussed only probabilistically regarding the fault model and scheduling, would it not be reasonable to relax the system dynamics accordingly?

The answer is: Yes, the system dynamics do not necessarily have to be deterministic. For instance, multiple guards can simultaneously be active and the choice which command is carried out could be probabilistic or non-deterministic. Yet, the goal of this thesis is to establish a method to evaluate the fault tolerance — with a focus on the dynamic aspects of recovery from transient faults — of system dynamics under a given probabilistic environment. To achieve this, a *simple* mode of reasoning — based on *deterministic* system dynamics — is applied first. An extension to probabilistic and non-deterministic system dynamics is discussed in the future work section in chapter 8.

3.1.3   Liveness

Selected definitions of liveness from literature are provided in appendix A.4.5. Informally, liveness means that "the good" happens eventually [Lamport, 1977]. In the context of finite branching structures, this means that "the good" happens *deterministically within finite time or finitely many computation steps*.

Definition 3.4 (Liveness)**.**
A system $\mathfrak{S}$ is live w.r.t. to an event when that event is guaranteed to occur eventually.

There are two types of events — the desired "good" things — considered in this thesis that both require liveness. The first is *algorithmic liveness*.

Definition 3.5 (Algorithmic liveness)**.**
An algorithm $\mathfrak{A}$ is live when it causes the system to change its state continually.

The desired event for algorithmic liveness is that the system changes its state, that it shows algorithmic progress of some kind. A system that is not algorithmically live is *silent*. It reaches a certain state eventually and does not change its state afterwards. For instance, the traffic light algorithm from the previous chapter is algorithmically live. The second liveness aspect is *recovery liveness*.

Definition 3.6 (Recovery liveness)**.**
The recovery of a system is live when the system progresses towards the legal set of states with regards to a ranking function.

The desired event for recovery liveness is to reach a legal state. Recovery liveness insinuates a ranking among the states regarding their distance to the legal set of states. With the discussion about the Hamming distance in mind, the reasonable ordering among system states is obvious: The legal states have a Hamming distance of 0. The illegal states from which any legal state can be reached with one computation step — considering the execution semantics — have a Hamming distance of 1 and so forth. The maximal Hamming distance has the state in which all registers are corrupted. It requires as many computation steps as there are processes in the system, considering every process has one register and serial execution semantics apply. Its Hamming distance then coincides with the number of processes. When the algorithm guarantees that in the absence of faults either

- a system *continuously* decreases its Hamming distance towards the next proximal legal state with every computation step (weak recovery liveness, i.e. continuous progress towards $\mathcal{S}_{legal}$) or

- a system continually decreases and never increases its Hamming distance towards the next proximal legal state (strong recovery liveness, i.e. continual progress towards $\mathcal{S}_{legal}$)

it provides for recovery liveness. Both algorithmic and recovery liveness are not in conflict. A system can provide recovery liveness and be functionally silent with regards to algorithmic liveness at the same time. That is, recovery and algorithmic behavior are mutually independent and define the whole set of behavior described by the algorithm.

Showing that both liveness types are mutually independent for execution traces also shows they are independent regarding the guarded commands. Since both guards and safety are state-based, each command can be attributed exclusively to either recovery or algorithmic liveness. Therefore, their independence is shown for the more general case of execution traces. Assume that safety is defined not state-based but over execution traces. Then, a guarded command can be enabled for a state that is safe in one trace and unsafe in another trace. Yet, no state can be satisfying and violating safety *at the same time*. The respective state still belongs to one partition exclusively at that time point and the recovery types are never intersecting. This discussion thread continues in paragraph "State-based safety" on page 33.

Commonly, systems (or more specifically algorithms) under probabilistic schedulers and serial execution semantics cannot accomplish to guarantee *weak* recovery liveness as one process that is required to execute to converge to a legal state might be postponed indefinitely. Yet, there are exceptions. The TLA for instance converges instantly to a legal state regardless which one of the two processes executes. Nevertheless, in most cases only strong recovery liveness is provided.

3.1.4 Threats

The threat branch provides the antagonist to the fault-tolerant system. Its application in this thesis is discussed in section 2.2.1. The threat branch classification is based on the work by Avižienis et al. [Avižienis et al., 2001, Avižienis et al., 2004]. Notably, we assume that faults only corrupt the registers of currently executing processes. The scheduler is immune to faults.

Definition 3.7 (Transient fault)**.**
A transient fault (or fault step) is a computation step that is not (necessarily[4]) conform with the algorithm. Its result is the executing process' register to possibly store an unintended value. A fault step does not cause the system state to violate safety. Faults are not detected by the system and tolerable.

Faults do not necessarily have an effect on the system behavior if they remain *undetected* until they vanish. A faulty state does not yet violate safety.

Definition 3.8 (Error)**.**
A fault becomes an error upon its detection. It is detectable by violating safety. An error remains an error as long as it is is tolerable and until it becomes intolerable or it vanishes.

An error is the stage at which the system state deviates from the legal states and at which that deviation is detected, but at which it is still *tolerable*. During the error stage, the system has the opportunity to recover. If the recovery takes *too long*, that is, when the system did not recover *in time*, the error becomes *intolerable*.

Definition 3.9 (Failure)**.**
A failure is an intolerable detected deviation from the intended system behavior. It occurs when safety conditions have been continuously violated for too long, when a system did not recover from an error in time.

---

[4]It is not conform for malign faults and conform for benign faults.

With a perfect fault detector, faults become errors instantly. When the system leaves no time for repair and must satisfy safety instantly upon request, errors become failures instantly. This thesis focuses on relaxing the latter to evaluate how time helps the system to avoid errors from becoming failures. This allows to determine recovery liveness, the speed with which a system recuperates from the effects of faults.

The *threat cycle* in figure 2.1 shows the transitions from legal states to subsequently fault, error and failure. Here, the system can recover from each threat stage. Notably, errors and failures are perceived as safety violations. Some authors consider safety violations as "irreversible" like Alpern [Alpern and Schneider, 1985] while others do not like Lamport [Lamport, 1977]. This thesis follows the latter perception to allow for recovery liveness.

In this thesis, the deterministic system dynamics — which are system topology and algorithm — are not prone to transient faults. The volatile memory, the process registers, on the other hand, are prone to transient faults. The scheduler is probabilistic like the fault model on the one hand, demonstrating the versatility of the approach, while being immune to faults on the other hand. Its immunity is justified in paragraph "Immunity of algorithm and scheduler to faults" on page 8. The deterministic system dynamics are supposed to provide for recovery liveness. Hence, the perception that safety violations can be treated by recovery is suitable in this context and the distinction into errors and failures allows to quantify the recovery of deterministic system dynamics in a probabilistic environment.

There are two important remarks differentiating the definitions in this thesis from other related work.

Components and system

The causal chain proposed by Avižienis et al. [Avižienis et al., 2004] distinguishes between fault, error and failure from two perspectives, component and system. When a component fails — referred to as *local* failure — the effects possibly propagate into the system, in which the local failure is perceived as a fault. In this thesis, the terms fault, error and failure are used solely in the system context to avoid confusion.

Are failures permanent?

One common interpretation of failure is the transition "from correct to incorrect service" [Avižienis et al., 2004]. When interpreted as *safety* violation, failures are "irremediable" [Alpern and Schneider, 1985]. As discussed for errors before, this thesis perceives safety violations, and thus also failures, as *remediable*. A fault is an undetected perturbation that becomes an error when it is detected. When detected, the error can be treated. During that process, the system *knows* that it does not work correctly, meaning that it does not meet its specified safety conditions during that time. It possibly can deprive its service from the system user during that time. In case the system does not recover *in time*, the error becomes a failure.

The transition from error to failure is — in the context of this thesis — purely time related. We focus on the relation between the amount of time for recovery and the probability that the system is in a *legal* state. The time-dependent transition from error to failure is important here.

3.1.5 Types and means of fault tolerance

One common classification of fault tolerance is based on the use of means that can be exploited to provide fault tolerance.

Means of fault tolerance

Detectors and correctors are such means. Detectors are related to safety. They promote faults to errors and failures. Correctors are related to liveness. They allow the system to recover from faults, errors and failures. When the system is allowed to temporarily violate safety, correctors allow to prevent the transition from error to failure for those execution traces that do not violate safety for too long. Notably, faults need not necessarily be detected to be correctable. For instance assume a transient fault perturbing a register and the register being overwritten before it is read. Then, the fault was corrected by overwriting it before it was detectable by reading it.

Types of fault tolerance

In a deterministic setting, when fault tolerance is discussed with regards to specific faults, the *combination of detectors and correctors* determines the fault tolerance *type*.

- **no detectors, no correctors:** When a system can neither detect nor correct specific faults, providing statements regarding its safety or liveness is not possible. In this context, the system is not fault-tolerant.

- **only detectors:** A system with detectors for specific faults can shut down to prevent a fault from becoming a failure[5]. As the system ensures safety, it is *failsafe* fault-tolerant regarding the faults it can detect and prevent from becoming errors. When the system fails, that is, as soon as the system state is about to violate safety conditions, it fails *safely*, actually *before* violating safety conditions. "The bad" (i.e. violating safety conditions) does not happen. Yet, when failing safely, the system stops operating, possibly violating liveness conditions. "The good" — which here is termination, reaching a legal state or simply being accessible — will then not happen. Informally, a failsafe system only provides correct answers or service or none at all.

- **only correctors:** Systems with correctors for specific faults continue to operate even in the presence of safety violations caused by these faults. While safety is violated, that is, when recovery liveness takes over from algorithmic liveness, the system user is exposed to the *unsafe* system behavior. Since the system user is exposed to that unsafe behavior, that is, when the faulty service cannot be temporarily deprived upon detection, this fault tolerance type is called *non-masking* fault tolerance. The effects of the faults are not *masked* from — or hidden from or transparent to — the system user. Non-masking fault-tolerant systems are continuously accessible. They always provide an answer/service. Yet, they are only continually *available*. The term *available* is explained in the next section.

---

[5] In that context, the system is shut down when the transition from fault to error is imminent.

- **both detectors and correctors:** Systems with both detectors and correctors can detect specific faults and deprive the system user until the correctors let the system recover to a legal state, assuming that both detectors and correctors have the same fault coverage. The effects of faults, that is, errors and failures, from that specific fault coverage are thus transparent to the system user, apart from a possible admissible delay that is required by recovery liveness. The system is *masking fault-tolerant* for these specific faults. It continuously satisfies safety and continuously provides liveness regarding these specific faults.

	continuously safe	continually safe
recovery liveness	masking	non-masking
no recovery liveness	failsafe	

Table 3.1: Defining fault tolerance types via fault tolerance properties

When proving fault-tolerance properties, the goal is to show that the effects of specific — in that sense deterministic — faults can be dealt with *maskingly*. Arora and Kulkarni [Arora and Kulkarni, 1998a, Arora and Kulkarni, 1998b, Kulkarni, 1999] provide related workin that regard. The scope of this thesis is yet the quantification of recovery under *probabilistic* conditions. This thesis targets the gap between non-masking and masking fault tolerance by evaluating recovery liveness. The transition from masking to non-masking fault tolerance is taken by those transitions that take *too long* for their recovery in a probabilistic environment.

Fault coverage

The term fault coverage has been defined for various contexts [Williams and Sunter, 2000]. In the context of fault tolerance, fault coverage means that a system is non-masking fault tolerant for one class of faults, failsafe fault tolerant for another class of faults, masking fault tolerant for the intersection of both classes and intolerant for all faults for which it is neither non-masking nor failsafe fault tolerant. The goal of this thesis is to exploit this classification, to extend it from *classifying faults* being specifically either non-masking, failsafe or masking tolerable. This thesis computes *how* fault-tolerant a system is with regards to each fault tolerance type. This discussion thread is continued in section 3.3.

3.1.6   Fault tolerance measures

Measures allow to quantify the fault tolerance of a system, that is, how well it provides for desired fault tolerance attributes like safety. This section defines the fault tolerance measures availability and reliability.

Availability

Selected definitions from literature are provided in appendix A.4.7.

Definition 3.10 (Availability)**.**
Availability is the mean probability that a system satisfies its safety conditions.

Let $MTTF$ be the *mean time to failure*, $MTTR$ be the *mean time to repair* and $MTBF = MTTF + MTTR$ be the *mean time between failures*. Then, availability is defined as follows:

$$A = MTTF/MTBF \tag{3.1}$$

In the context of the *threat cycle* shown in figure 2.1, errors instantly advance to failures. Yet, to measure how well recovery can be exploited if the system is allowed to recover from the effects of faults *before* errors become failures, a more distinguished approach is required. To measure the recovery *over time*, availability must be measured with regards to time.

Definition 3.11 (Point availability)**.**
The point availability is the probability that the system is in a state satisfying safety at that time point: $A_t = \sum_{s_i \in \mathcal{S}_{legal}} pr(s_i)_t$

The point availability is the aggregated probability that the system is in any legal state. As discussed in paragraph "Types of fault tolerance" on page 31, this thesis focuses on masking and non-masking fault-tolerant systems that can recover from transient faults. These systems are commonly designed to run indefinitely. Hence, availability of a system *in the limit* must be defined.

Definition 3.12 (Limiting availability)**.**
The limiting availability is the limiting value of $A(t)$ *as* t *approaches infinity, if existent.*

Reliability

While availability is the measure for masking and non-masking fault-tolerant systems, reliability is appropriate for failsafe systems.

Definition 3.13 (Reliability)**.**
The reliability of a system with respect to time point t *is the probability that the system continuously satisfies its safety conditions until that time point.*

Accordingly, the *limiting reliability* with a probabilistic fault model is 0 [Trivedi, 2002, p.321] as it continuously decreases under a probabilistic fault model. Informally, reliability with regards to a time point t is the probability that the system survives until that time point.

State-based safety

This paragraph returns to recovery and algorithmic liveness to motivate the definition of safety being defined state-based. The safety predicate partitions the state space into legal and illegal states. The algorithm specifies *operations* within both partitions: recovery operations in the illegal states and desired service operations in the legal states. It might yet be desirable to specify safety based on execution traces within the legal states. Contrary to state-based safety, the notion of *operationality*[6] allows to define safety accordingly.

This thesis focuses on i) transient faults putting a system in recovery mode and ii) the convergence dynamics of that recovery. The *correct* execution, that is, the system being *operational*, in absence of transient faults is not addressed. While availability and reliability measure the transition probabilities between both legal and and illegal partitions, operationality is suitable to measure algorithmic liveness within the legal states. Hence, contrary to operationality, safety is defined purely state-based in this thesis.

---

[6]The term has been coined by Keller in 1987 [Keller, 1987].

3.1.7   Redundancy

Means of fault tolerance commonly utilize redundancy. Popular examples are error detecting and correcting codes utilizing spatial redundancy to implement parity bits based on generator polynomials on the one hand and time to compute and check these polynomials on the other hand. This thesis focuses on the time-based recovery dynamics. Spatial redundant fault tolerance mechanisms like parity bits in registers are considered to be part of the system under investigation.Regarding redundancy, the question driving this thesis is: How far can the availability of a non-masking fault-tolerant system be increased for an amount of temporal redundancy?

3.2   Self-stabilization

Self-stabilization is a suitable concept to reason about the recovery of non-masking and masking fault-tolerant systems introduced by Dijkstra in 1974 [Dijkstra, 1974]. Notably, it consists of two deterministic properties as pointed out in the definition by Schneider in 1993, which is based on the classic definition:

Definition 3.14 (Self stabilization - Schneider [Schneider, 1993, p.3])**.**
We define self-stabilization for a system[7] $\mathfrak{S}$ with respect to a predicate $\mathcal{P}$, over its set of global states, where $\mathcal{P}$ is intended to identify its correct execution. $\mathfrak{S}$ is self-stabilizing with respect to predicate $\mathcal{P}$ if it satisfies the following two properties:

- **Closure:** *$\mathcal{P}$ is closed under the execution of $\mathfrak{S}$. That is, once $\mathcal{P}$ is established in $\mathfrak{S}$, it cannot be falsified.*

- **Convergence:** *Starting from an arbitrary global state, $\mathfrak{S}$ is guaranteed to reach a global state satisfying $\mathcal{P}$ within a finite number of state transitions.*

Notably, convergence must succeed "within a finite number of state transitions", regardless of whether the state space is finite or infinite. This attribute of convergence is often referred to as "eventually" [Dolev, 2000].

Probabilistic self-stabilization

As the thesis focuses on recovery liveness, the convergence property is important. Algorithmic liveness is related to the closure property. With probabilistic scheduling, *weak* recovery liveness cannot be guaranteed as the scheduler can ignore a process that is required to execute to complete convergence. Hence, only *strong* recovery liveness can be achieved. Devismes et al. [Devismes et al., 2008] provide a suitable extension to self-stabilization in 2008:

Definition 3.15 (Probabilistic self-stabilization - Devismes et al. [Devismes et al., 2008])**.**

$\mathfrak{S}$ is[8] probabilistically self-stabilizing for $\mathcal{P}$ if there exists a non-empty subset of $\mathcal{S}$, noted $\mathcal{S}_{legal}$, such that: (i) Any execution of $\mathfrak{S}$ starting from a configuration of $\mathcal{S}_{legal}$ always satisfies $\mathcal{P}$ (Strong Closure Property), and (ii) Starting from any configuration, any execution of $\mathfrak{S}$ reaches a configuration of $\mathcal{S}_{legal}$ with Probability 1 (Probabilistic Convergence Property).

---

[7]The symbols have been adapted to suit this thesis.
[8]The symbols have been adapted to suit this thesis.

What does convergence mean? Dijkstra and Schneider define convergence such that the legal set of states is reached "in finite time". Convergence does not mean that the distance to the legal states is continuously or continually decreased. Probabilistic convergence replaces the term *eventually* by *with probability 1*. Thus, probabilistic self-stabilization fits perfectly with the context discussed in section 2.3[9].

Convergence vs. recovery liveness

Recovery liveness means progress towards the set of legal states. Regarding the progress towards the set of legal states, probabilistic convergence is a super-set of recovery liveness as it also allows to temporarily recede from (i.e. increase the Hamming distance to) the set of legal states. This means that — regarding the progress towards the set of legal states — every system providing weak recovery liveness also provides strong recovery liveness, and that every system providing strong recovery liveness provides also probabilistic convergence as shown in figure 3.4.

Yet, neither weak nor strong recovery liveness hold for convergence, as continuous or continual progress towards the set of legal states does not imply ever reaching them. Assume a continuous state space or a state space with infinitely many states. Then, the system might continuously or continually approach the set of legal states without ever reaching them. One example execution trace is an inward bound spiral that approaches the center point arbitrarily close without ever reaching it. It would satisfy recovery liveness but not (probabilistic) convergence. Yet, in the context of finite discrete state spaces as applied in this thesis, recovery liveness does imply probabilistic convergence.
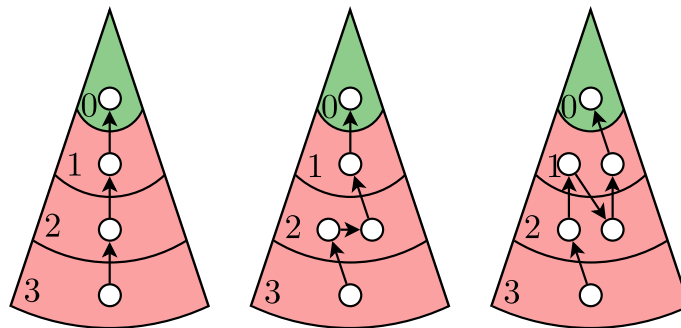
Figure 3.4: Execution traces permitted by weak recovery liveness (left), strong recovery liveness (middle) and (probabilistic) convergence (right) examples. The sectors are labeled according to the Hamming distance.

The examples presented in this thesis provide for both probabilistic convergence and strong recovery liveness. The benefit of strong recovery liveness is that it allows to easily show probabilistic convergence. The methods proposed in this thesis yet hold for probabilistic self-stabilizing systems in general.

Non-masking fault tolerance and self-stabilization

Self-stabilizing systems are non-masking fault tolerant, but not every non-masking fault tolerant system is self-stabilizing. The concepts and methods discussed in this thesis

---

[9]See paragraph before "Abbreviations" on page 13.

generally apply to non-masking fault tolerant systems. Yet the examples show only self-stabilizing systems. The motivation behind that is twofold.

First, self-stabilizing systems are deterministically designed to cope with the effects of transient faults. This makes it comfortable to distinguish between the effects of transient faults and non-deterministic or probabilistic system design[10]. The second benefit is difficult to grasp at this stage as the means to understand it are discussed in the following chapters. Stabilization is a concept that works similar to fault propagation. Opposed to sporadic faults, it provides deterministic *control* via the algorithm to assure that the system converges to a legal state. The processes in a self-stabilizing system *communicate*. They cooperate according to the algorithm to allow for convergence. Analyzing fault tolerance properties of *uncontrolled* non-masking fault tolerant systems, in which the processes neither propagate the effects of faults nor cooperate to allow for self-stabilization, is *simple* compared to *controlled* processes. Section 7.1 later provides a coherent case study to explain this argument in detail.

3.3   Design for masking fault tolerance

As previously discussed, Arora and Kulkarni [Arora and Kulkarni, 1998a, Arora and Kulkarni, 1998b, Kulkarni, 1999] provide the formalisms and concepts to discuss fault tolerance design. Contrary to the probabilistic perspective of this thesis, they focus on deterministically satisfying fault tolerance types with respect to specific *fault coverages*. In fault tolerance design, an intolerant system is subsequently amended by detectors and then by correctors to acquire a functional equivalent yet fault-tolerant system. The following figure shows how means can be combined to achieve masking fault tolerance with respect to specific faults.

$$\text{intolerant} \xrightarrow{\text{correctors}} \text{non}-\text{masking}$$
$$\downarrow \text{detectors} \qquad\qquad\qquad \downarrow \text{detectors}$$
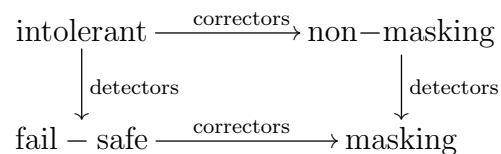$$\text{fail} - \text{safe} \xrightarrow{\text{correctors}} \text{masking}$$

Figure 3.5: From fault intolerance to masking fault tolerance

A system without detectors and correctors, for which no assertions about its fault tolerance can be made, becomes non-masking fault-tolerant against faults when correctors are added correcting these faults. It becomes failsafe fault-tolerant against faults when detectors are added detecting these faults. A non-masking fault-tolerant system becomes masking fault-tolerant against faults when detectors are added detecting the faults also covered by the corrector. A failsafe fault-tolerant system becomes masking fault-tolerant against faults when correctors are added correcting the faults also covered by the detector. Notably, a system is only masking fault-tolerant against faults in the cut-set of the detector's and corrector's fault coverages as depicted in the following figure 3.6.

---

[10]At least in the context of this thesis, distinguishing between the effects of faults and non-deterministic or probabilistic algorithms is important, as explained in section 3.5.

Figure 3.6: Fault tolerance classes

The effects of faults that are detected and corrected, which is the green area in the above figure, are masked. Effects of faults that are neither detected nor corrected belong to the red area and are, in the context of fault tolerance, *not supported*. But what does this classification mean for a system user and how would probabilistic dynamics of recovery fit in?

Fault tolerance type from the user perspective

Assume a user accessing a system. Means of fault tolerance like detectors and correctors are supplied as a layer between user and system. The user has a set of requests to the system. At each time step, the user *requests* the system service and expects a correct response the very same time step. The user requests are queued and the user provides them as a sequence. A request is repeated when an incorrect answer is detected. The following figure 3.7 depicts the setting:
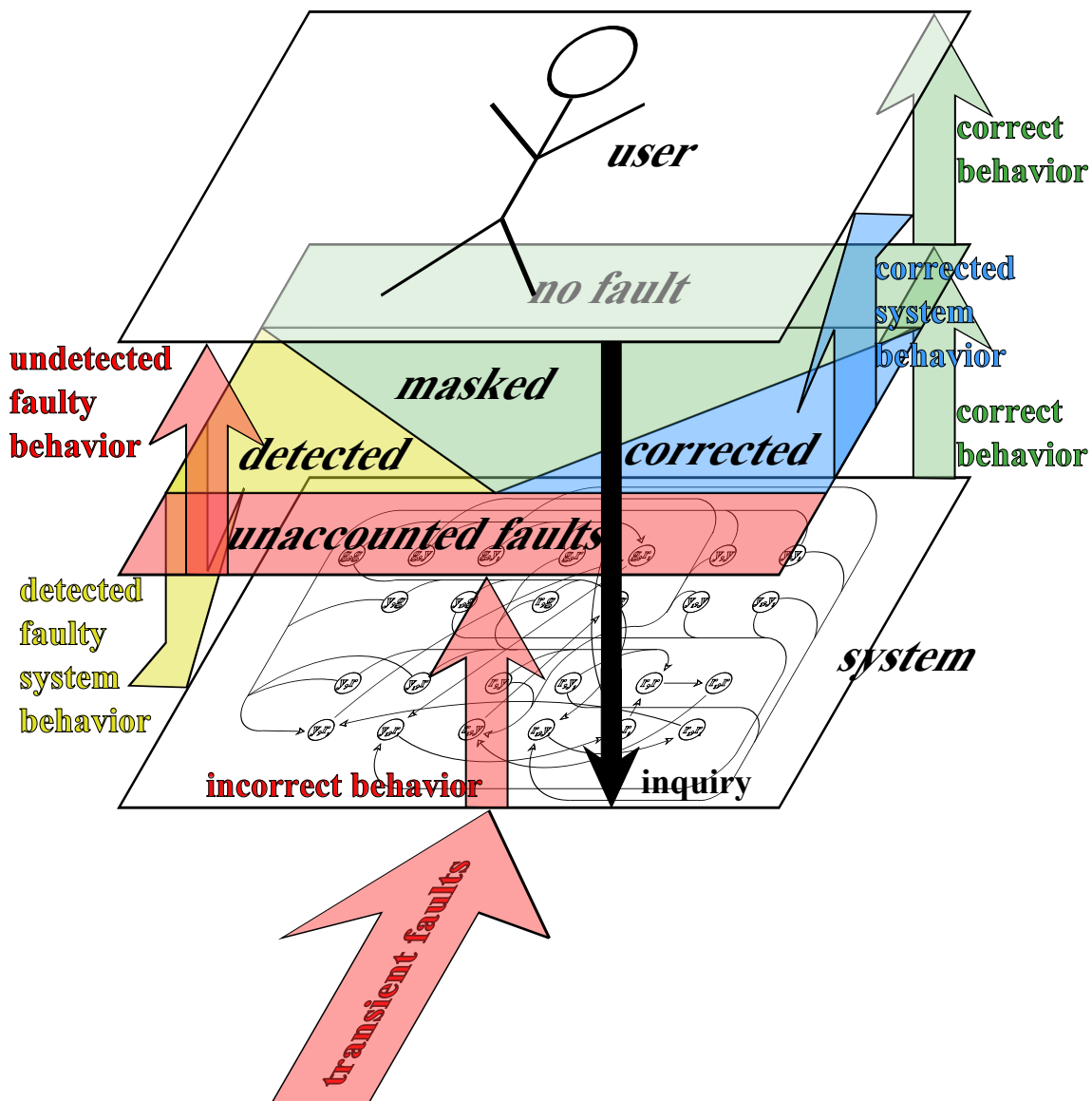
Figure 3.7: System behavior

A system user (top layer) requests some service from a distributed system (lower layer) depicted as bold black downward arrow. Transient faults influence the system. Between the two layers is a fault tolerance layer comprising detectors (*yellow*) and correctors (*blue*) that overlap (*green*). While posting requests is assumed to be immune to faults[11], the responses computed by the system[12] are prone to transient faults occurring in the system. Responses are depicted as colored upward arrows.

The fault tolerance layer is designed analogous to the fault tolerance classes shown in figure 3.6. A system response is either carried out without malign fault (correct behavior, lower arrow), or it provides a wrong answer (incorrect behavior). The fault tolerance layer takes the system response and applies detectors and correctors. In case no fault is detected in a correct answer (correct behavior, upper arrow) or *in situ* correction is applicable

---

[11]This assumption is covered by the goal of this thesis being the fault tolerance of the underlying system and not of the transmission media.

[12]Again, the transmission itself is assumed to be immune.

(corrected system behavior), the correct answer is provided to the user. In case a fault is not detected, the user is exposed to its effects (undetected faulty behavior). In case a fault is detected but cannot be corrected, the request is returned to the system (detected faulty system behavior). Notably, detectors might malfunction, too. These malfunctions are called false positives and false negatives. The scheduler and the algorithm are not pictured here.

Figure 3.7 shows how fault tolerance can be added to a system without manipulating the system itself. Three changes to this model are necessary to fit the scope of this thesis. The first is, that aspects of *in situ* correction are not the topic of this thesis and are thus excluded. Second, detector malfunctions are excluded. The goal to quantify the fault tolerance of the system and not the fault tolerance of the detectors. Third, the fault coverage classification must be adapted to account for the recovery dynamics of the system. These adaptations are discussed in the following section.

3.4 Fault tolerance configurations

In the beginning of this chapter, figure 3.1 painted the picture of a user interacting with a system and a detection layer between them. In this section, this picture is continued. The user is now permanently requesting the system service. In the optimal case there are no effects of faults present in the system and the detection layer passes correct responses directly to the user. But how are temporal constraints modeled? What about *unreliable* detectors that raise the detection flag wrongfully with no actual fault present?

The discussion in section 3.1.5 motivated the focus on non-masking fault-tolerant systems that can utilize time to withhold incorrect service from the user for a limited amount of time. The three fundamental questions in that context are:

> **Unmasking fault tolerance** (Fundamental questions)**.**
>
> a) Is the system in a safe state?
>
> b) Is an error detected?
>
> c) Are temporal constraints regarding convergence violated?

Configurations

All three questions can be answered with either "yes" or "no" at each time point. For brevity, "yes" is encoded with 1 and "no" with 0. The configurations are:

$\langle a), b), c)\rangle$	meaning
$\langle 1 , 0 , 0 \rangle$	The system is in a legal state, no fault is detected and temporal constraints are not violated.
$\langle 1 , 1 , 0 \rangle$	The system is in a legal state, yet a fault is detected (false positive) but temporal constraints are not violated yet.
$\langle 1 , 1 , 1 \rangle$	The system is in a legal state, yet a fault is detected (false positive) and the detection flag has been raised for too long.
$\langle 0 , 1 , 0 \rangle$	A fault is correctly detected and the correction does not yet violate temporal constraints.
$\langle 0 , 1 , 1 \rangle$	A fault occurred and was detected, but a legal state could not be reached within time.
$\langle 0 , 0 , 0 \rangle$	An undetected fault causes the system to deliver wrong results and persists until it is either detected or corrected by chance.

Table 3.2: Fault tolerance configurations

The first digit in every triple answers the first question, the second digit the second question and so forth. Two configurations are omitted: $\langle 0, 0, 1 \rangle$ and $\langle 1, 0, 1 \rangle$. We assume that the system service is deprived from the system user when the detection flag is raised, regardless if an actual error is present or not. With the detection flag not being raised, the system is not deprived from the user and temporal constraints are not violated. The two states can thus be omitted. Notably, the questions still distinguish between correctly and incorrectly detected faults, the so-called *false positives*.

Transitions between configurations

Configuration $\langle 1, 0, 0 \rangle$ (green, center) is the desired predicate combination. The system provides a correct answer and no fault is detected. If detectors trigger a false alarm (false positive), the system converges to configuration $\langle 1, 1, 0 \rangle$ (yellow, upper right).
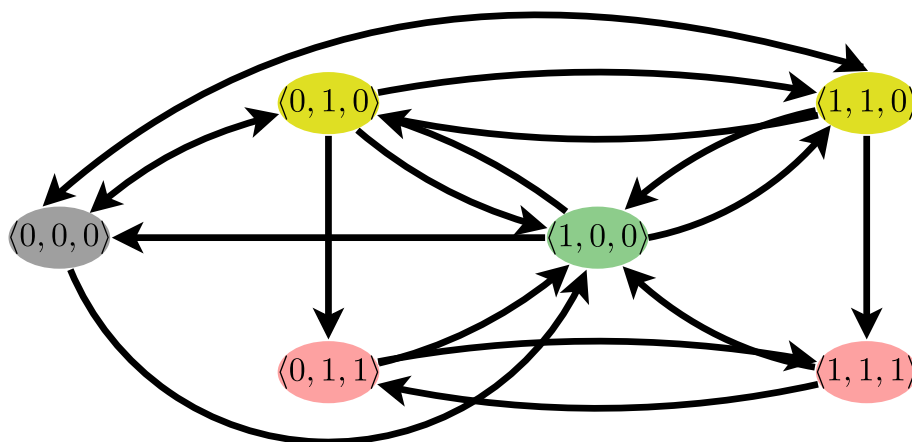

Figure 3.8: Configuration transition diagram

Else, if detectors trigger an alarm correctly, the system converges to configuration $\langle 0, 1, 0 \rangle$ (yellow, upper left). In both (yellow) cases, the system retries the last inquiry until it either succeeds (towards green, or gray in case of an undetected fault) or until temporal

constraints are violated (red). The amount of time that the system is granted to succeed is the amount time that the user is willing to wait. Else, after temporal constraints are violated, the system reaches the particular lower configuration, depending whether the fault was detected correctly $\langle 0, 1, 1 \rangle$ (red, lower left) or not $\langle 1, 1, 1 \rangle$ (red, lower right). Even after temporal constraints are violated, the system can recover to a legal state. Finally, there is also the possibility that faults are and remain undetected (false negative) and the user is unknowingly exposed to an *incorrect* service modeled by configuration $\langle 0, 0, 0 \rangle$ (gray). Regarding the *fault coverage*, an insufficient fault model is the common cause for false positives. In this final case, either the fault is eventually detected, or the fault is washed out prior to its detection. It might also occur that a persisting fault is temporarily detected while not violating temporal constraints (i.e. the transitions between configurations $\langle 0, 1, 0 \rangle$ and $\langle 1, 1, 0 \rangle$), or that a persisting fault is temporarily undetected while violating temporal constraints (i.e. the transitions between configurations $\langle 0, 1, 1 \rangle$ and $\langle 1, 1, 1 \rangle$).

Bounded recovery liveness

In the above configurations, the transitions from the yellow to the red states are taken when temporal constraints regarding the recovery are violated, when converging to the legal states *took too long*. These transitions coincide with the transition from error to failure as discussed in the threat cycle on page 10. We define bounded liveness with regards to a maximal admissible recovery time window to address this property formally:

Definition 3.16 (Bounded recovery liveness)**.**
Let w be the maximal admissible amount of time (here: computation steps) allowed to complete convergence, hereafter referred to as recovery time window. A partial execution trace $\sigma_{t,k}^i$ is bounded recovery live w.r.t. w, if it does not continuously (i.e. without interruption) raise the detection flag for that duration within the trace.

If the recovery time window w exceeds the length of the partial execution trace k, that partial execution trace satisfies bounded recovery liveness. Otherwise, the system must complete convergence within the partial execution trace. Notably, fault bursts might continually perturb the system such that bounded recovery liveness is violated.

3.5 Unmasking fault tolerance

A deterministically masking fault-tolerant system guarantees to complete convergence within at most w computation steps for specific faults. With a probabilistic environment, such a guarantee is obsolete. The goal in optimizing the fault tolerance of a system is to minimize the probability of the system to stay in an unsafe state for more than w steps. To find the system (design) that offers the best chance to complete convergence, the recovery liveness of the system must be measured. The recovery liveness of the system is the transition $\overrightarrow{\langle 0, 1, 0 \rangle, \langle 1, 0, 0 \rangle}$ in figure 3.8. To focus on that transitions, this section prunes all transitions that are not required in this context.

The fault masker

To measure the recovery liveness of the system with regards to a probabilistic environment, false positives and negatives must be excluded. This thesis considers the layer of *detection to be perfect* in order to exclude false positives as well as false negatives, represented by the right yellow and red configurations and the gray configuration. A perfect

detector never wrongfully raises the detection flag (i.e. the second digit in the configuration transition diagram). A perfect detector detects *every* fault and *instantly* promotes it to an error. The latter part of the assumption is a simplification, assuming that the amount of time consumed by detection is negligible compared to the amount of time that is consumed by one computation step executed by the system. Detection is thereby assumed to occur within each computation step.

Assume a perfect fault detector allowing for instantaneous fault detection of all faults and faults only [Müllner et al., 2009, p.63]. Thereby, faults are instantly promoted to errors. For this reason, faults and errors are concluded in figure 2.1. We refer to a perfect fault detector as *fault masker*. It forces the system to retry inquiries with erroneous system responses. If the demand is not satisfied within the recovery time window w, the error becomes a failure as shown in figure 3.9(a). Otherwise, the effects of faults are masked as shown in figure 3.9(b).

(a) Fault masker fails   (b) Fault masker succeeds
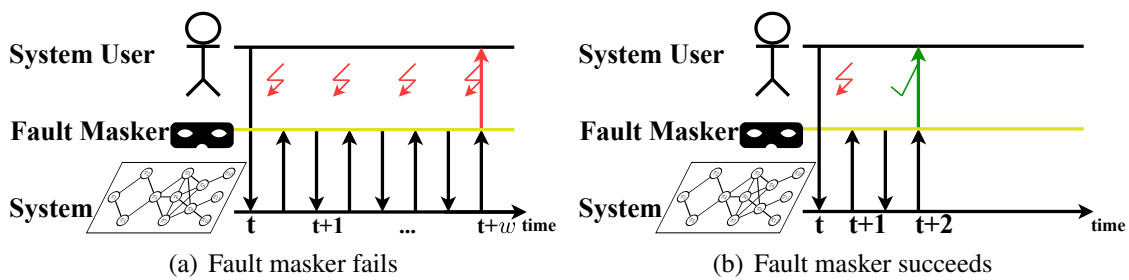
Figure 3.9: The fault masker

The fault masker prunes the configurations that are redundant for the evaluation of recovery liveness. The resulting diagram, shown in figure 3.10 on the left hand side, coincides with the threat cycle introduced in figure 2.1 shown on the right hand side. It reduces the configuration transition diagram from figure 3.8 to those configurations that are required to measure the fault tolerance of a system.
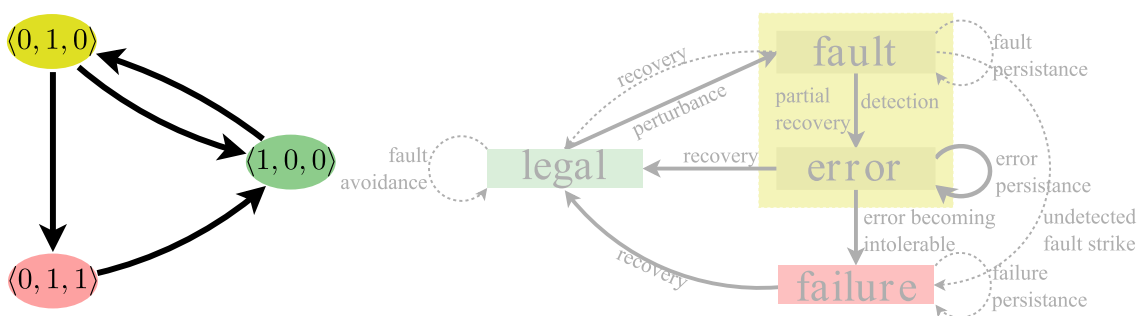
Figure 3.10: Reduced configuration transition diagram, perfect detectors

The fault masker allows to focus on quantifying the recovery dynamics of the system. It filters out the effects of faulty detectors to solely account for the quality of the correctors. For sake of completeness, the following paragraph reasons about what happens when perfect correctors are assumed and faulty detectors are employed before the next chapter introduces a fault tolerance measure for the quantification.

Untrusting fault tolerance

Consider a set of applicable detectors of varying quality and a perfect corrector. Every detected error is corrected instantaneously. The original model from figure 3.8 is then reduced to the configurations and transitions shown in figure 3.11. No (detected) error is promoted to become a failure. The configurations $\langle 0, 1, 0 \rangle$ and $\langle 1, 1, 0 \rangle$ are traversed instantaneously, that is, *in situ*, indicated as dotted arrows. Under perfect correction, configuration $\langle 1, 1, 0 \rangle$ leaves room for interpretation. How does perfect correction work when there is no error present? We assume that correction will correct the *non-fault* and return immediately (after one step) to configuration $\langle 1, 0, 0 \rangle$ like it does for configuration $\langle 0, 1, 0 \rangle$. The intricate part of the discussion is configuration $\langle 0, 0, 0 \rangle$. With no error detected, temporal constraints cannot be violated. The type of non-masking tolerance exposes the system user to the effects of undetected faults only.
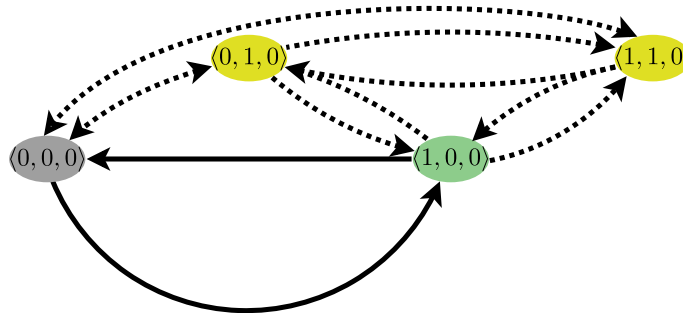

Figure 3.11: Reduced Configuration Transition Diagram, perfect correctors

Ergodicity of the configuration transition diagram

Consider a system user continuously requesting the system service with some finite potential to wait and a non-masking fault tolerant system shielded by a fault masker and exposed to probabilistic transient faults as described above. The corresponding reduced configuration transition diagram is ergodic. The goal of the following chapters is to derive a relation, mapping the system's transition model onto the reduced configuration transition model. The challenge is to account for state safety and temporal constraints. The following chapter introduces a fault tolerance measure that is suitable for measuring the reduced configuration transition diagram and shows how the DTMC of a system can be adapted to compute that measure.

3.6 Summarizing fault tolerance terminology and taxonomy

This chapter introduced a fault tolerance taxonomy that is suitable for the scope of this thesis and, in the same context, provided definitions for the terminology. The concept of self-stabilization was discussed and the focus on the probabilistic variant was motivated. The *design for masking fault tolerance* paradigm by Arora and Kulkarni was discussed and adapted to suit a probabilistic context. The concept of the fault masker was introduced to prune those configurations that are not required for quantifying recovery dynamics. The next chapter builds on this formal background and introduces a novel fault tolerance measure to quantify the recovery dynamics.

4. Limiting window availability

This chapter introduces the fault tolerance measure *limiting window availability* (LWA) and presents a general method to compute it. LWA quantifies the recovery dynamics in the limit as discussed in the previous chapter. Parts of this chapter are published [Müllner and Theel, 2011, Müllner et al., 2012, Müllner et al., 2013].

Motivating LWA

Quantifying the recovery dynamics of a non-masking fault-tolerant system, whose service can be deprived while errors are present, allows to compare different solutions to the same problem regarding their efficiency in exploiting temporal redundancy for fault tolerance. Similar to optimal generator polynomials in the domain of spatial redundancy, specific non-masking fault-tolerant designs have characteristic optimal offsets in the trade-off between the amount of temporal redundancy and recovery probability. When the maximal admissible amount of time for recovery is known, those system designs are optimal that have an offset closest to but smaller than that maximal admissible amount of time. Otherwise, when saving time is a secondary objective and the recovery probability must not be below a certain threshold, the system designs can be ordered according to the time they require to achieve the desired probability. The most economic system achieving that probability is then optimal. Thus, LWA is a valuable indicator that allows to compare systems according to their ability to recover from transient faults.

The history of the term limiting windows availability

At the beginning of the studies for this thesis, the idea was to measure the availability of a system and to determine its recovery. In case a system is unavailable, how does its

availability change when the user is willing to wait for some time? This coined the term *window availability*. Initially, in 2009 [Müllner et al., 2009], the setup comprised a system that was initially unavailable and executing a fixed number of computation steps before its availability was measured, thus being defined as *instantaneous window availability*. With the discussion that indefinitely running systems converge to the stationary distribution, it was motivated to assume that this distribution should hold at the time a user requests the system service, which is *in the limit*. Hence, the recovery of non-masking fault tolerant systems was measured with *limiting window availability*.

Structure of this chapter

Section 4.1 contains the formal definition of LWA. Section 4.2 explains how LWA is computed. LWA is specifically defined to measure the fault tolerance of non-masking fault-tolerant systems under the fault masker. The design decisions for LWA are discussed in the following section along with possible alterations like limiting window reliability. After the general method to compute the LWA is introduced and the design decisions are motivated, section 4.3 shows on three examples how LWA can be computed. The evaluation, interpretation and comparison of solutions are discussed in section 4.5.

4.1   Defining limiting window availability

Consider a system $\mathfrak{S}$ executing a self-stabilizing algorithm being exposed to a probabilistic fault environment. A fault masker is mounted between the system and its user. LWA of that system with regards to a specific amount of time — the time window for recovery — is the probability for the system to having reached a safe state at least once within that time window, considering that the initial probability for the system to be in a certain state coincides with the stationary distribution. In case the effects of all present faults are eliminated within the time window, failures do not arise. In such cases, the effects of faults are successfully masked within the given time window. Otherwise, if the repair takes too long, errors become failures. Then, the effects of faults could not have been masked within the given time window. In that case, the system user is either provided with the corrupted value, with an error message, or both, depending on the system design and fault tolerance type. The first option (corrupted value) is in accordance with the non-masking fault tolerance type and the second option allows for the design of failsafe fault-tolerant systems (although the information needs not necessarily be exploited). The third option is reasonable when *degraded* corrupted values decrease the functionality, but allow the system to maintain a lower level of service or operation.

Regarding this thesis' context, the first option is selected to solve the question, how far faults can be contained and treated within a given amount of time, and how far errors become failures. From the perspective of fault tolerance types, the question translates to: how masking is an otherwise non-masking system if it is provided with a fault masker and a limited amount of time to stabilize? The amount of time that a system should be allowed for recovery needs not necessarily to be predetermined. Therefore, we refer to that maximal admissible amount of time as *time window* which can be arbitrarily wide opened[1]. The width of the time window — which is the duration for recovery or allowed convalescence — is addressed as parameter w. When w is infinite, probabilistic convergence is achieved. This thesis focuses on finite values for w.

---

[1] The notion of *interval availability*, cf. Appendix A.4.7, is similar.

As motivated in the previous chapter, an ergodic transition model is assumed and furthermore that the system user accesses the system after it converged to its stationary distribution $Pr_\Omega(\mathcal{S})$. The limiting window availability of window size w, labeled l_w, is the probability with which the system is available to the user for at least one computation step within w computation steps.

There are multiple approaches to formalize LWA. This thesis discusses two of these approaches, the first being easier to understand, the second one being more precise. LWA can be formalized as shown in equation 4.1.

$$l_w = pr(\exists i \in [\Omega, \Omega + w] : s_i \models \mathcal{P}) \tag{4.1}$$

LWA of window size w is the probability that there exists a legal state within the corresponding time window. A more precise but harder to understand approach is to define LWA via execution traces and the first hitting time[2]. The first hitting time in this context is the time step at which the trajectory first reaches a legal state.

Definition 4.1 (Limiting Window Availability).
A system $\mathfrak{S}$ executing a (probabilistic) self-stabilizing algorithm is under a probabilistic influence. Its corresponding transition model $\mathcal{D}$ is in its stationary distribution $Pr_\Omega(\mathcal{S})$. Let

$$T_s : \inf_{t>0}\{s_t \in \mathcal{S}_{legal}|s_0 = s\} \tag{4.2}$$

be the first time the system reaches a legal state for each execution trace. The limiting window availability of window size w, denoted as l_w, is the probability that the system functions correctly with regards to a safety predicate $\mathcal{P}$ at least once within that window:

$$l_w = \sum_{s\in\mathcal{S}} pr(T_s \leq w \wedge s_0 = s) \cdot pr_\Omega(s) \tag{4.3}$$

LWA of time window w is the accumulated probability mass of all partial execution traces of interval length w that reach the legal set of states — meaning they contain at least one legal state — with $Pr(\mathcal{S})_0 := Pr(\mathcal{S})_\Omega$. The *limiting availability* $\sum_{s_i\in\mathcal{S}_{legal}} pr_0(s_i)$ coincides with l_0. LWA of a time window $w = 1$, which is l_1, is the probability that the system is either initially in a safe state or, in case it was initially not in a safe state, that it is in a safe state one time step later. The trajectories in which the system state is legal at both time points is covered by the first case. By increasing the window, the probability for the system to successfully recover eventually, increases, too. LWA is an accumulated distribution function, a probability measure on stopping times. It assigns a probability mass for each stopping time at which the system probably reaches a legal state. LWA in the context of *probabilistic real time computational tree logic* (PCTL) is discussed in the future work section in chapter 8.

Absorbing states

The stationary probability distribution $Pr_\Omega(\mathcal{S})$ assigns probability mass to each state in which the system can possibly be in in the limit. With each further computation step, the

---

[2]The lower case letter τ that is commonly used to refer to the stopping or Markov time is reserved as splitting operator in chapter 6. Hence, the upper case letter T is used instead here.

set of partial execution traces $\sigma_{\Omega,\Omega+w}$ that reach a safe state for the first time after the limit, grows. Hence, the aggregated probability continuously increases with w. The set of legal states *absorbs* those traces hitting the legal set of states for the first time. Even in case the system is perturbed by a fault again afterwards in an execution trace, the goal of reaching the set of legal states would have been accomplished. The set of legal states within the otherwise ergodic DTMC $\mathcal{D}$ becomes *absorbing* when computing LWA.

Hansson and Jonsson [Hansson and Jonsson, 1994] provide a similar approach based on an extension of the computational tree logic (CTL) as introduced by Emerson et al. [Clarke et al., 1986]. They also exploit DTMCs and focus on algorithms to verify if desired conditions — specified in *probabilistic real time CTL* (PCTL) — hold. In that context, LWA can be expressed with

$$L(P(\diamond^{\leq w} s \models \mathcal{P})) \geq pr \tag{4.4}$$

Although their provided methods are closely related, the nature of their work is different. While they introduce a general logic, this thesis focuses on one distinct predicate. The nature of their work is to provide general algorithms for checking DTMCs and to reason about their complexity. This thesis on the other hand aims at reducing the complexity of checking DTMCs specifically in the context of quantifying fault tolerance measures. Although quantifying fault tolerance measures and *probabilistic real time CTL* share a common ground, the contribution of this thesis lies not in exploiting *probabilistic real time CTL*, but in finding a notion of time-restricted fault tolerance and its quantification. The exploitation and application of methods that have been introduced in a general context is discussed in the future work section in chapter 8.

4.1.1 Limiting window availability vector

While the LWA l_w is a point availability, we are interested in the sequence of these probabilities over time. Such a LWA vector v can be either finite, bounded by a finite w, or infinite with $w = \infty$.

Definition 4.2 (Limiting window availability vector)**.**
The LWA vector v is a (finite or infinite) vector of probabilities

$$v = \langle l_0, l_1, \ldots \rangle \tag{4.5}$$

such that $\forall l_i, l_j : 0 < l_i, l_j \leq 1$, and $\forall i < j; i, j \in \mathbb{N}_0 : l_i < l_j$.

Notably, $0 < l_i$ or else $\mathcal{D}$ would not be ergodic[3], and $l_i \leq 1$ since $l_\infty = 1$.

Estimating a reasonable window size

There are two motivations to set a fixed maximal admissible window size. Typically, either safety specifications constrain the maximal admissible window size (e.g. a *point of no return*), or the window size has to be increased until a specific probability mass is reached. In the first case, w is simply set to that maximal admissible window size and the aggregated probability mass l_w is computed. In the second case, w is successively increased until l_w exceeds the minimal required availability for the first time. When the desired minimal required availability is smaller than one, it is achieved within finite time.

---

[3]Further possibilities are i) another initial probability distribution and ii) an empty set of legal states. The first case is discussed in paragraph "An Exception to Strict Monotonicity" in this section.

4.1.2 Limiting window availability vector gradient

In case of neither the system specification requiring temporal boundaries (i.e. starvation is not an issue), nor a fixed demand requiring a minimal desired availability, a third possible motivation to compute the LWA might be to determine the *sweet spot*. At this point in time, the probability increase to reach a safe state is maximal. The *gradient* of the LWA vector shows the increase in probability mass for each additional time step spent. It can be used as an indicator to determine the *sweet spot*. The question for the sweet spot asks:

At which time step is the increase of probability to reach a safe state maximal?

Definition 4.3 (Limiting window availability vector gradient)**.**
The v gradient (or LWA vector gradient / differential), denoted as g, is a finite or infinite vector such that

$$g = \langle g_1, \ldots, g_i \rangle = \langle l_1 - l_0, \ldots, l_i - l_{i-1}, \ldots \rangle \qquad (4.6)$$

with $|v| - 1 = |g|$

4.1.3 Instantaneous window availability

This section discusses a variation of LWA to demonstrate the versatility of the notion of *window availability* in general, and to exploit the benefits of the variation to discuss monotonicity of the vector gradient. The LWA increases strictly monotonic over time. With an ergodic DTMC, each state contains probability mass in the limit, including states with a Hamming distance of 1. Furthermore, the transition probabilities from these states to a legal state are positive. Hence, the probability mass in the legal states strictly monotonically increases. One of the basic design decisions in defining LWA was to set $Pr_0(\mathcal{S}) := Pr_\Omega(\mathcal{S})$. For other probability distributions, for instance when the initial states all have a Hamming distance of 2, the property of *strict* monotonicity would not necessarily hold. Yet, regular monotonicity holds as long as there are states within the set of initial states with a Hamming distance smaller w.

For the following example we assume the worst case that the system is initially in a state with the maximal Hamming distance. Hence, definitions 4.1, 4.2 and 4.3 do not apply here. This example stems from a comparison between fault tolerance evaluation by simulation and by analysis [Müllner et al., 2009]. The experiment was conducted on a self-stabilizing BFS algorithm [Dolev, 2000] executing on a four process topology with $E = \{(\pi_1, \pi_4), (\pi_2, \pi_3), (\pi_2, \pi_4), (\pi_3, \pi_4)\}$ (cf. figure A.5 in the appendix on page 166). The processes execute under serial execution semantics and a probabilistic scheduler. The registers are exposed to probabilistic transient faults. As a variation to LWA, the system is initially deterministically in the state in which every register is corrupted. Therefore, the fault tolerance measure computed is not the *limiting* window availability but the *instantaneous* window availability (IWA) with a lead tie of zero and the presented initial state. This deviation is motivated for two reasons: it provides an example for which *strict* monotonicity does not hold and it *amplifies* the vector differential as there is no probability mass not available to the recovery. The system requires at least four computation steps under serial execution semantics to reach the set of legal states. Figure 4.1 shows the LWA vector gradient g for that system for different fault probabilities $q \in \{0.01, 0.03, 0.06, 0.08, 0.1\}$.
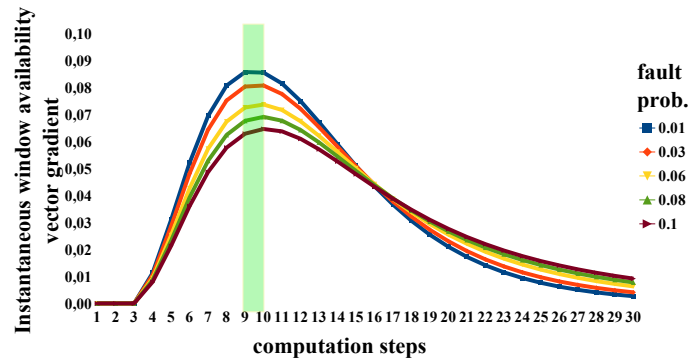
Figure 4.1: Instantaneous window availability gradient - analysis via PRISM [Kwiatkowska et al., 2002]

The *sweet spot* in this example is clearly between the ninth and the tenth time step regardless of the fault probability. After that, the probability mass increase is reduced. The example also demonstrates how available tools — like PRISM[4] [Kwiatkowska et al., 2002] in this case — can be exploited. A direct comparison between the results for the four process topology between PRISM and simulation can be found in [Müllner et al., 2009]. One of the challenges with the analysis with PRISM — concerning the available computing power in 2009 — was that *larger* systems soon rendered intractable with the system size increasing. Therefore, a simulation based approach was proposed. Section 7.2 provides a further example computing the IWA in the context of the decomposition-and-lumping-approach presented in chapters 5 and 6.

Sample-based analysis via simulation

As discussed before, computing the LWA suffers from state space explosion. One way to avoid this issue is to consider only a limited amount of execution traces by restricting the analysis to sampling-based methods like simulation. An example similar to the previous one was conducted on a larger topology that was not tractable with PRISM and the available computing power back then. The example topology comprised eight processes (cf. figure A.6 on page 166) executing the same BFS algorithm. The experiment consisted of ten trials, one for each fault probability. Each trial was executed one million times and the time span until the set of legal states was first reached was counted. Each time, the initial state was selected randomly and the lead tie was set to 1000. The results are shown in figure 4.2.

---

[4]The PRISM source code for this example is available online http://www.informatik.uni-oldenburg.de/~phoenix/docs/UFirst09_IWA.rar.
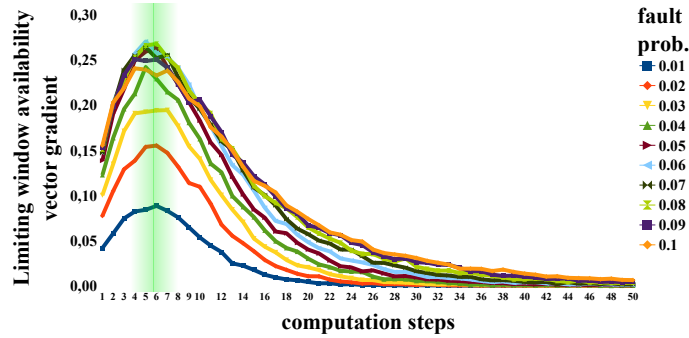
Figure 4.2: Limiting window availability gradient - simulation via SiSSDA [Müllner, 2007]

Comparing the LWA vector gradients of both four and eight process experiments indicates that i) the shape of the graph is typical for this setting and that ii) simulation is a viable means for locating the region in which the sweet spot is located in case a system is intractably large for the analysis. While the analysis provides precise results, depicted as solid green block in figure 4.1, the simulation based approach indicates the region of the sweet spot only with a certain *confidence* as indicated by the green waveform. Furthermore, the initial probability for the fault probabilities is different, showing that the trajectories in figure 4.1 are in opposite order until computation step 15 as compared to the order in figure 4.2.

4.2 Computing limiting window availability

This section informally describes how LWA of a system under a given environment can be computed. Given the system specifications and a fault model, an ergodic DTMC $\mathcal{D}$ can be derived as discussed in the previous chapter. Being ergodic, the stationary distribution Pr_Ω of $\mathcal{D}$ can be computed. To determine LWA (and v and g), the bounded reachability, which is the probability mass *leaking* from the set of illegal states into the set of legal states over time, is calculated by making the legal states absorbing. When the system reaches a legal state, the user inquiry succeeds. The adapted DTMC with absorbing legal states is labeled $\mathcal{D}_{\text{LWA}}$. In $\mathcal{D}_{\text{LWA}}$, no probability mass emanates from the absorbing legal states towards the illegal states. Let $Pr_\Omega(\mathcal{S})$ be the stationary distribution of $\mathcal{D}$. The probability distribution for each following time step is computed with $\forall i > 0 : Pr_{\Omega+i}(\mathcal{S}) = \mathcal{D}_{\text{LWA}} \cdot Pr_{\Omega+1-i}(\mathcal{S})$. Thereby, the accumulated probability mass for each time step in the absorbing legal states can be computed, thus calculating the LWA vector. The complexity of computing the LWA vector is thus linear in the maximal window size to be determined.

4.3 Examples

This section provides three examples. The first example in section 4.3.1 serves only to motivate LWA. The traffic lights example in section 4.3.2 then catches up with the TLA to show how LWA of a small distributed system comprising only two processes can be computed. Section 4.3.3 then introduces the self-stabilizing broadcast algorithm (BASS). Compared to the TLA, the BASS is simple (only three guarded commands compared to

50 in the TLA, and only three possible variable allocations instead of five), making it attractive to discuss the analysis of larger systems (i.e. more processes) and to investigate the impact of fault propagation.

4.3.1   Motivational example

Although the rather formal concept of LWA and its related entities might seem abstract, it already is anticipated it in our everyday lives. For instance when an internet browser (i.e. the machine running it) is disconnected from the internet, the browser will throw an appropriate error message after some time. What the annoyed user does not see is the fact that until the error message is displayed, the browser automatically (re-)tries to reach the requested website several times. In case each one of the connection attempts fails, the browser quickly realizes that further attempts are futile. Otherwise, in case the browser receives at least partially correct information (or any information at all), it will invest further retries. In that case, it takes longer to surrender. Although the target is not unreachable in the latter case, the browser will usually tell so. The requested site is just not reachable *enough*. It takes the browser more retries and thereby longer to determine that the probability to ultimately succeed is sufficiently low to throw an error message compared to the case where it has no connection at all.

4.3.2   Self-stabilizing traffic lights algorithm (TLA)

This section continues the traffic lights example from chapter 2. We specify the safety predicate as follows:

$$s_i \models \mathcal{P} \Leftrightarrow R_1 = red \vee R_1 = red_1 \vee R_2 = red \vee R_2 = red_1 \tag{4.7}$$

At least one of the traffic lights must show red. Then, at most one party can have access and there cannot occur a crash. The safety predicate partitions the state space from figure 2.5 into legal and illegal states as shown in figure 4.3.
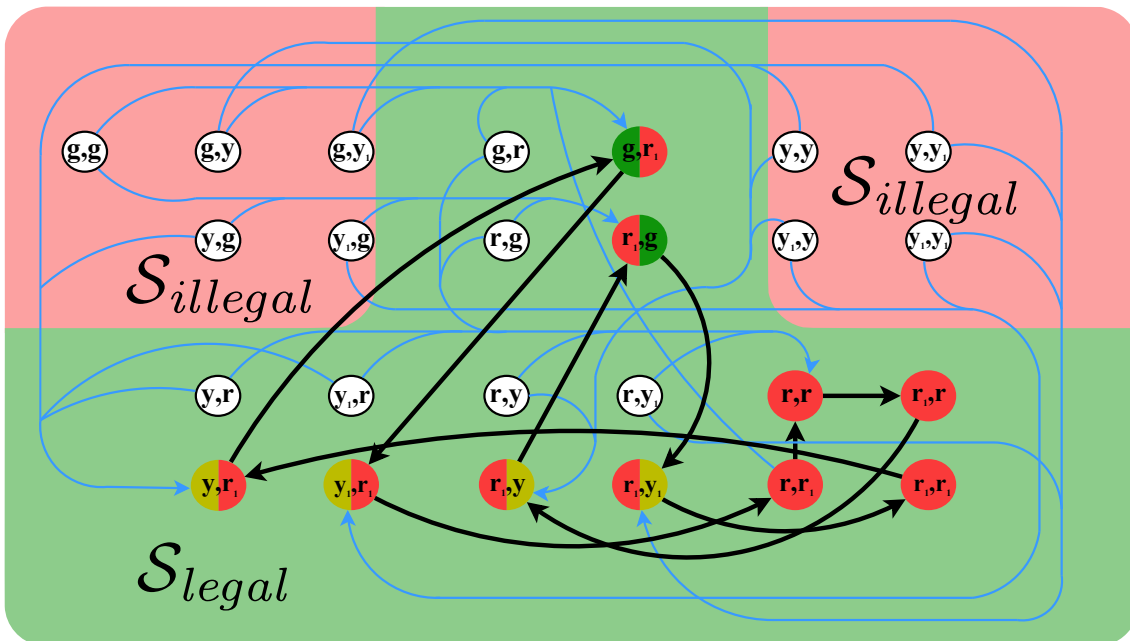

Figure 4.3: State space partitioning via predicate $\mathcal{P}$

Algorithm 2.1 contains the guarded commands providing algorithmic as well as strong recovery liveness. Since the system provides for strong recovery liveness, it guarantees recovery with probability 1. We assume that eternity is not the amount of time a system user would be willing to wait.

Despite the possibility for a symbolic computation of LWA, the actual probabilities are computed as presented in paragraph "Fault Model" on page 20. The numerical transition matrix shown in table 4.1 is computed.

Part 1

↓ from/to →	g,g	g,y	g,y_1	y,g	y_1,g	g,r	g,r_1	r,g	r_1,g	y,y	y,y_1	y_1,y	y_1,y_1
g,g	0.05	0.025	0.025	0.025	0.025	0.025	0.4	0.025	0.4				
g,y	0.025	0.05	0.025			0.025	0.4				0.025	0.025	
g,y_1	0.025	0.025	0.05			0.025	0.4					0.025	0.025
y,g	0.025			0.05	0.025			0.025	0.4	0.025	0.025		
y_1,g	0.025			0.025	0.05			0.025	0.4			0.025	0.025
g,r	0.025	0.025	0.025			0.05	0.4						
g,r_1	0.025	0.025	0.025			0.025	0.425						
r,g	0.025			0.025	0.025			0.05	0.4				
r_1,g	0.025			0.025	0.025			0.025	0.425				
y,y		0.025		0.025						0.05	0.025	0.025	
y,y_1			0.025	0.025						0.025	0.05		0.025
y_1,y		0.025			0.025					0.025		0.05	0.025
y_1,y_1			0.025		0.025						0.025	0.025	0.05
y,r				0.025		0.025					0.025	0.025	
y_1,r					0.025	0.025						0.025	0.025
y,r_1				0.025			0.4				0.025	0.025	
y_1,r_1					0.025		0.025					0.025	0.025
r,y		0.025						0.025		0.025		0.025	
r,y_1			0.025					0.025			0.025		0.025
r_1,y		0.025							0.4	0.025		0.025	
r_1,y_1			0.025						0.025		0.025		0.025
r,r						0.025		0.025					
r_1,r						0.025			0.025				
r,r_1							0.025	0.025					
r_1,r_1							0.025		0.4				

Part 2

↓ from/to →	y,r	y_1,r	y,r_1	y_1,r_1	r,y	r,y_1	r_1,y	r_1,y_1	r,r	r_1,r	r,r_1	r_1,r_1
g,y					0.025		0.4					
g,y_1						0.025		0.4				
y,g	0.025		0.4									
y_1,g		0.025		0.4								
g,r	0.025	0.025							0.4	0.025		
g,r_1			0.025	0.4							0.025	0.025
r,g					0.025	0.025			0.4	0.025		0.025
r_1,g							0.025	0.4		0.025		0.025
y,y	0.025		0.4		0.025		0.4					
y,y_1	0.025		0.4			0.025		0.4				
y_1,y		0.025		0.4	0.025		0.4					
y_1,y_1		0.025		0.4		0.025		0.4				
y,r	0.05	0.025	0.4						0.4	0.025		
y_1,r	0.025	0.05		0.4					0.4	0.025		
y,r_1	0.025		0.425	0.025							0.025	0.025
y_1,r_1		0.025	0.025	0.425							0.4	0.025
r,y					0.05	0.025	0.4		0.4		0.025	
r,y_1					0.025	0.05		0.4	0.4		0.025	
r_1,y					0.025		0.425	0.025		0.025		0.025
r_1,y_1						0.025	0.025	0.425		0.025		0.4
r,r	0.025	0.025			0.025	0.025			0.425	0.4	0.025	
r_1,r	0.025	0.025					0.4	0.025	0.025	0.425		0.025
r,r_1			0.025	0.025	0.025	0.025			0.4		0.425	0.025
r_1,r_1			0.4	0.025			0.025	0.025			0.025	0.05

Table 4.1: Transition matrix of the ergodic DTMC $\mathcal{D}$ of TLA with numerical values

Computing the stationary probability distribution is demonstrated on an example in MatLab in appendix A.5.2 on page 166.

state	stationary probability	state	stationary probability	state	stationary probability
$\langle g,g\rangle$	0.006833008158440	$\langle g,y\rangle$	0.005419916453587	$\langle g,y_1\rangle$	0.006496008792927
$\langle g,r\rangle$	0.007384644613641	$\langle g,r_1\rangle$	0.080896038928274	$\langle r,g\rangle$	0.008754549072939
$\langle y,y_1\rangle$	0.006361104635544	$\langle y_1,y\rangle$	0.005510976759820	$\langle y_1,y_1\rangle$	0.006587069099161
$\langle y,r_1\rangle$	0.076039489262492	$\langle y_1,r_1\rangle$	0.084174209952678	$\langle r,y\rangle$	0.007341457368085
$\langle r_1,y_1\rangle$	0.124949002535158	$\langle r,r\rangle$	0.086556689346863	$\langle r_1,r\rangle$	0.079689388262128

state	stationary probability	state	stationary probability
$\langle y,g\rangle$	0.006698104001057	$\langle y_1,g\rangle$	0.006924068464673
$\langle r_1,g\rangle$	0.137080979693621	$\langle y,y\rangle$	0.005285012296204
$\langle y,r\rangle$	0.007249740456258	$\langle y_1,r\rangle$	0.007475704919874
$\langle r,y_1\rangle$	0.008417549707426	$\langle r_1,y\rangle$	0.086209678318900
$\langle r,r_1\rangle$	0.072821008031510	$\langle r_1,r_1\rangle$	0.068844600868740

Table 4.2: Stationary probability distribution $Pr_\Omega(\mathcal{S})$ of $\mathcal{D}(\mathcal{S} \times \mathcal{S})$

Predicating desired properties

The traffic lights example demonstrates that not only safety, but a variety of desired properties can be identified. Two desired properties are obvious:

- safety (cf. definition 3.6 and predicate 4.7) and

- operability (cf. sequence 2.3):

$$s_{i,t} \models \mathcal{P}_{op} \Leftrightarrow s_{i,t} \in \{\langle g,r_1\rangle, \langle r,g\rangle, \langle r,r\rangle, \langle r_1,r\rangle, \langle y,r_1\rangle,$$
$$\langle y_1,r_1\rangle, \langle r_1,y\rangle, \langle r_1,y_1\rangle, \langle r,r_1\rangle, \langle r_1,r_1\rangle\} \tag{4.8}$$

Liveness and bounded liveness predicates can be defined analogously via (bounded) execution traces. There is a marginal difference between the two predicates $\mathcal{P}$ and $\mathcal{P}_{op}$. While $s_{i,t} \models \mathcal{P}_{op}$ means that the system is in a desired state, $s_{i,t} \models \mathcal{P}_{safe}$ means that the system is not in an undesired state. The difference is made by six states that neither violate safety nor satisfy operability. The cardinalities[5] are $|\mathcal{P}| = 16$ and $|\mathcal{P}_{op}| = 10$. This shows that the analysis is not necessarily restricted to measuring *safety*, but that it is possible to analyze any desired property that can be likewise formalized as a predicate. To compute LWA, only safety is regarded.

Remark 4.1 (Limiting availability).
The limiting availability (cf. Appendix A.4.7) is the aggregated probability mass of those states that are considered to be safe, which in the current case of the TLA example is:

$$A_\infty(\mathfrak{S}) = l_0 = \sum_{s_i \in \mathcal{S}_{legal}} pr_\Omega(s_i) = 0.943884731338587 \tag{4.9}$$

To compute LWA, all legal states of the DTMC become absorbing states as discussed in the corresponding paragraph on page 47. Hence, all self-targeting transitions that originate from a legal state are set to probability 1, while all other transitions originating from a legal state that are not self-targeting are set to probability 0.

---

[5]We abbreviate $|s \models \mathcal{P}|$, the number of legal states, with $|\mathcal{P}|$ and analogously for all other predicates.

↓ from/to →	g,g	g,y	g,y_1	y,g	y_1,g	g,r	g,r_1	r,g	r_1,g	y,y	y,y_1	y_1,y	y_1,y_1
g,g	0.05	0.025	0.025	0.025	0.025	0.025	0.4	0.025	0.4				
g,y	0.025	0.05	0.025			0.025	0.4			0.025		0.025	
g,y_1	0.025	0.025	0.05			0.025	0.4				0.025		0.025
y,g	0.025			0.05	0.025			0.025	0.4	0.025	0.025		
y_1,g	0.025			0.025	0.05			0.025	0.4			0.025	0.025
g,r						1							
g,r_1							1						
r,g								1					
r_1,g									1				
y,y		0.025		0.025						0.05	0.025	0.025	
y,y_1			0.025	0.025						0.025	0.05		0.025
y_1,y		0.025			0.025					0.025		0.05	0.025
y_1,y_1			0.025		0.025						0.025	0.025	0.05

↓ from/to →	y,r	y_1,r	y,r_1	y_1,r_1	r,y	r,y_1	r_1,y	r_1,y_1	r,r	r_1,r	r,r_1	r_1,r_1
g,y					0.025		0.4					
g,y_1						0.025		0.4				
y,g	0.025		0.4									
y_1,g		0.025		0.4								
y,y	0.025		0.4		0.025		0.4					
y,y_1	0.025		0.4			0.025		0.4				
y_1,y		0.025		0.4	0.025		0.4					
y_1,y_1		0.025		0.4		0.025		0.4				
y,r	1											
y_1,r		1										
y,r_1			1									
y_1,r_1				1								
r,y					1							
r,y_1						1						
r_1,y							1					
r_1,y_1								1				
r,r									1			
r_1,r										1		
r,r_1											1	
r_1,r_1												1

Table 4.3: Transition matrix $Pr(\mathcal{S} \times \mathcal{S})$ of DTMC $\mathcal{D}_{\mathrm{LWA}}$ of the traffic lights example

With the stationary distribution set as initial probability distribution, LWA can easily be computed. The MatLab source code is provided in appendix A.5.6 on page 171.
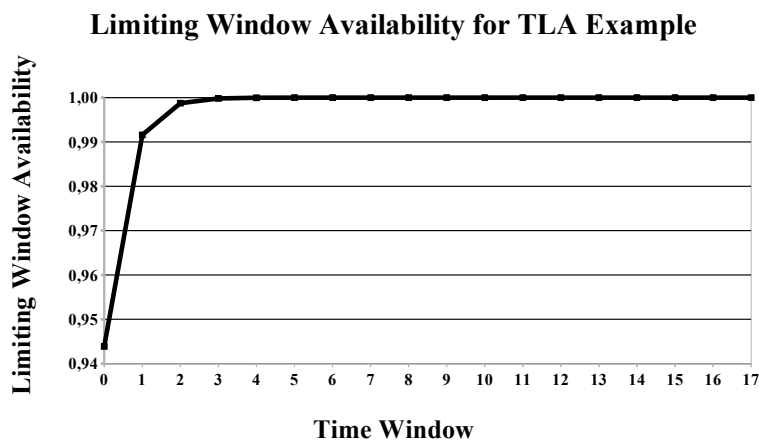

Figure 4.4: LWA of the traffic lights example

Figure 4.4 shows the probability mass within the legal states as it aggregates over time for the first 17 steps. Assume that the driver or walker observes the traffic light for some time steps during their approach to validate the accountability of the traffic light. The figure shows that even if the traffic lights violate safety specifications (in the limit), they converge fast. After two time steps, the probability for the traffic lights to have reached a legal state is above 0.99. After at most 17 steps, the aggregated probability mass reaches 1.000000000000000 with an accuracy of 15 decimal digits.

In engineering disciplines, the availability[6] of non-terminating systems is often given in terms of *nines*. For instance, an availability of five nines means 99.999%, which in downtime per year computes to $0.00001 \cdot 525960$ minutes $= 5.2596$ minutes downtime per year. The switch AXD301 by Ericsson for instance has an even lower average downtime of 0.631 seconds per year [Armstrong, 2007] and even undercuts nine nines: $\frac{0.631s}{1a} < 0.000000001$. Assume an average availability of nine nines is desired for the TLA example. LWA reaches that amount of probability with the eleventh step.

LWA can further be exploited to depict the probability mass for each state individually as it develops over time as shown in figure 4.5. The states are in the same order as in table 4.3. The figure shows how the illegal states are drained off their probability mass, converging to 0, while the 16 legal states gain probability mass. This helps in identifying illegal states withholding probability mass for too long. The according exploitation of this data is later exemplarily demonstrated on a larger example is section 6.5.2 on page 101.
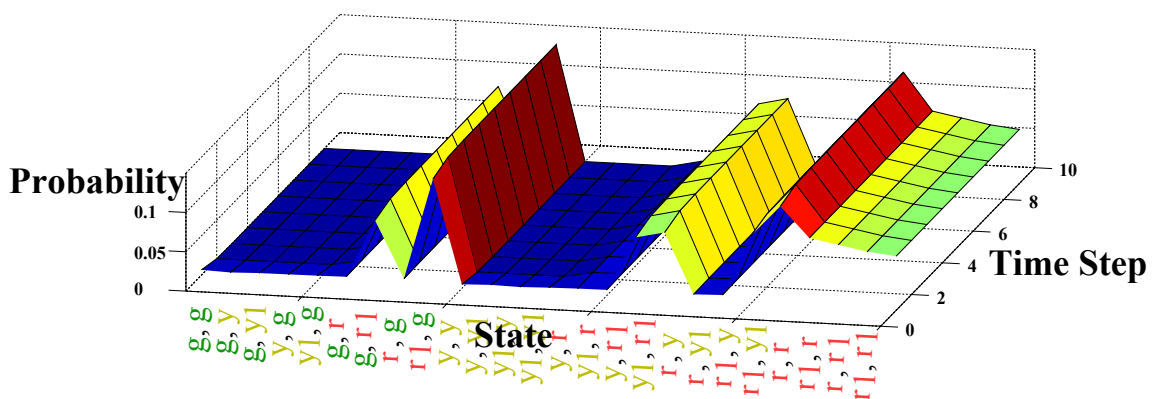

Figure 4.5: Probability distribution over states and time for five steps

Ruthless transition pruning

In table 4.3, all self-targeting transitions originating from legal states were set to 1. This procedure is valid for measuring LWA which is the aggregated probability mass of all legal states. For a more sophisticated analysis, in which the probability mass is evaluated individually for each state, such a simplification is not appropriate. Then, the transition probabilities originating from legal states and targeting illegal states must be set to 0 and the remaining transitions adapted accordingly.

One motivation to wait with the aggregation of the probability mass within state partitions is for instance the *predicate relaxation*. One might consider states where both traffic lights show a yellow sign at the same time *less critical* than states with one yellow and one green light, or both lights being green. Then, it is reasonable to consider the probability mass progress for each state separately to determine reasonable state combination permutations, for both relaxations as well as intensification, afterwards.

4.3.3   Self-stabilizing broadcast algorithm (BASS)

This example discusses the possibilities and challenges that come with hierarchical systems. Their simplicity regarding system topology, register domains and

---

[6]As stated in chapter 2, fault tolerance terminology is not consistent. Instead of availability, some sources refer to the average downtime of a system as reliability, cf. e.g. [**?**, p.199].

algorithm, predestines them for this discussion. Parts of this section are published [Müllner and Theel, 2011, Müllner et al., 2012, Müllner et al., 2013]. The goal of the BASS is to communicate a certain value among all processes from one designated root process to all other processes.

(Probabilistic) self-stabilization

Consider the algorithm shown in figure 4.6 (cf. [Müllner and Theel, 2011]). To increase readability, guarded commands are replaced by pseudo-code. Contrary to the TLA in which both processes depended on each other, this algorithm requires one designated process, referred to as *root* process, which is labeled π_1 by default. The root process is independent as it does not compute its own value based on the values in registers of other processes. All other *non-root* processes rely on the values stored in registers of processes that are closer to the root process than themselves. In this example, they even rely only on those neighbors that are *closest* to the root among their neighbors. Thereby, the processes executing BASS rely on each other *hierarchically*. An example topology is introduced in the following paragraph. The sub-algorithm executed by the root process is shown in algorithm 4.1 and the algorithm executed by all other processes is shown in algorithm 4.2.

const **id** := 0,
var R,
repeat {
   $R := 0$
}.

const **neighbors** := $\langle \pi_i, \ldots \rangle$,
const **distance** := min(**distance**(**neighbors**))+1,
const **set** := $\langle R_j, \ldots \rangle | \forall \pi_j :$
     $(\pi_j \in$ **neighbors**$) \wedge ($**distance**$(\pi_j) =$**distance**$-1)$,
var R,
repeat{
   $\neg((\exists R_i : \pi_i \in set \wedge R_i = 2)xor$
     $\exists R_i : \pi_i \in set \wedge R_i = 0))$
     $\rightarrow R := 1;$
  $\Box \exists R_i : \pi_i \in set \wedge R_i = 0$
     $\rightarrow R := 0;$
  $\Box \exists R_i : \pi_i \in set \wedge R_i = 2$
     $\rightarrow R := 2$
}.

Algorithm 4.1: Root Process   **Algorithm 4.2:** Non-root Processes

Figure 4.6: Self-stabilizing broadcast algorithm (BASS)

The $\Box$ symbol in the algorithm demarcates an case block. In the repeat loop in algorithm 4.2, the register is set to 1 in case the first clause holds. Otherwise, if the second clause holds, it is set to 0. In any other case the third clause holds and the executing process sets its register to 2 .

As a canonical simplification, each process contains one register with a domain of three different values. A process π_i stores one of the three possible values (0, 1, 2) in its single register R_i. If $R_i = 0$ applies, then process π_i fulfills its *local* part in satisfying safety; R_i then currently satisfies its safety predicate $\mathcal{P}$, denoted by $R_i \models \mathcal{P}$. Safety is satisfied *globally* when all registers satisfy safety. Notably, in this scenario, registers do not mutually depend on each other to satisfy safety.

When the values stored in a register are not conform with the predicate, $R_i \not\models \mathcal{P}$, then π_i either *knowingly* cannot determine the correct value, for instance due to dependencies on unavailable data from other processes, or it is *unknowingly* perturbed by a fault, either directly or via (hierarchical) fault propagation. In the abstraction, R_i takes the value 1 when π_i *knowingly* cannot compute a correct value for its register, and $R_i := 2$ in case π_i *unknowingly* stores an incorrect[7] value. A system state $s_{i,t} = \langle R_{1,t}, \ldots \rangle$ satisfies the global safety predicate when all registers store 0:

$$s_{i,t} \models \mathcal{P} \Leftrightarrow \forall R_j \in s_{i,t} : R_i = 0 \tag{4.10}$$

The system topology

The system topology is shown in figure 4.7. It comprises seven processes $\Pi = \{\pi_1, \ldots, \pi_7\}$ such that $E = \{e_{1,2}, e_{1,3}, e_{2,4}, e_{3,4}, e_{4,5}, e_{4,6}, e_{5,7}, e_{6,7}\}$. As discussed in paragraph "Restricting communication via guards" on page 7, the algorithm utilizes the communication channels only unidirectionally as indicated by the arrows.
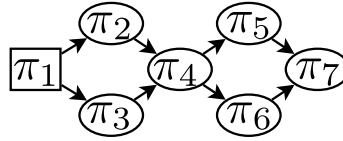

Figure 4.7: System

Furthermore, a probabilistic scheduler randomly selects the processes to execute under serial execution semantics.

The functionality

The root process π_1, when executing a computation step, stores the value 0 in its register in absence of a fault, and 2 if it is perturbed by a fault. Processes π_2 and π_3, when executing, copy the value of R_1 to their respective register. In case a process is not perturbed by a fault directly, it is possibly provided with contradicting data. In the example topology in figure 4.7, this is not possible for processes π_1, π_2 and π_3. For instance, when a process reads both 0 and 2 from the processes in its **set** (cf. line 3 in algorithm 4.2), then it writes 1 to its register. The value 1 means *don't know*. Otherwise, an undecidable process would have to make a non-deterministic or probabilistic choice between the values provided. Then, the algorithm is not self-stabilizing anymore as a process could always make the *wrong* decision, thereby preventing convergence. The 1 value provides clarity and prevents non-determinism.

- π_4 stores 0 when $(R_2 = 0 \wedge R_3 = 0) \vee (R_2 = 0 \wedge R_3 = 1) \vee (R_2 = 1 \wedge R_3 = 0)$.

- It stores 2 when $(R_2 = 2 \wedge R_3 = 2) \vee (R_2 = 2 \wedge R_3 = 1) \vee (R_2 = 1 \wedge R_3 = 2)$.

- The value 1 is stored otherwise, when both 0 and 2 are read.

---

[7]The abstraction does not distinguish between different faults. So if a process in the abstraction reads 2 twice, the abstraction pessimistically assumes that it might also read consistent values (e.g. originating from the same fault) in the concrete, and consequently accepts that value as locally correct.

Process π_7 executes the same way with respect to R_5 and R_6 (extending the third point, it stores 1 also when it only reads 1 from all processes in its **set**). Processes π_5 and π_6, when executing a computation step, adopt the value from R_4 to their respective register. The proposed algorithm is self-stabilizing with regards to $\mathcal{P}$.

Theorem 4.1 (The broadcast algorithm is self-stabilizing)**.**
The broadcast algorithm in figure 4.6 is self-stabilizing under a fair scheduler[8] and serial execution semantics for systems with finitely many processes.

Proof 4.1 (The broadcast algorithm is self-stabilizing)**.**
Anchor: The root process executes eventually writing 0 into its register.

Step: Eventually, every non-root process $\pi_i, 1 < i \leq n$ executes after all processes that are closer to the root than itself executed in the order of the path. Then, π_i writes 0 to its register.

Closing: Every process eventually stores 0 and no process can store a different value in absence of faults (closure). The algorithm is self-stabilizing. □

The algorithm is *silent* self-stabilizing [Dolev et al., 1996] as the system does not change its state once it completed convergence, meaning it is not algorithmically live. Theorem 4.1 and Proof 4.1 are published similarly in [Müllner and Theel, 2011, sec. 4.1] and have been adapted to suit this thesis. The system is *probabilistically* self-stabilizing under a probabilistic scheduler [Tixeuil, 2009, Devismes et al., 2008]. The proof is analogous to the proof of self-stabilization. The root process and following the non-root processes execute in descending order of their Hamming distance without being perturbed by errors with probability 1, thereby providing probabilistic convergence. The closure property remains untouched by the scheduler and is thereby also provided. The next step is to construct the ergodic transition model.

From system and environment models to the transition model

The state space contains the following states:

$$s_i \in \mathcal{S} : s_i = \langle R_1, R_2, \ldots, R_7 \rangle \in \{\langle 0,0,0,0,0,0,0 \rangle, \ldots, \langle 2,2,2,2,2,2,2 \rangle\} \quad (4.11)$$

The state space[9] comprises $2^3 \cdot 3^4 = 648$ states. Transient faults perturb the executing process with a probability $q = 1 - p$. The registers of non-executing processes remain untouched by faults. We select $q := 0.05$. A process stores 2 with a probability of 5% and executes as specified by the algorithm with a probability of 95%. The contour plot of the ergodic DTMC displayed in figure 4.8 shows the transition pattern. Each blue dot is a positive transition probability. The self-targeting transitions are on the diagonal from top left to bottom right.

---

[8]This excludes *probabilistically* fair schedulers.
[9]Processes π_1, π_2, and π_3 cannot derive 1. Therefore, only two different values (0 and 2) can be stored in the first three registers.
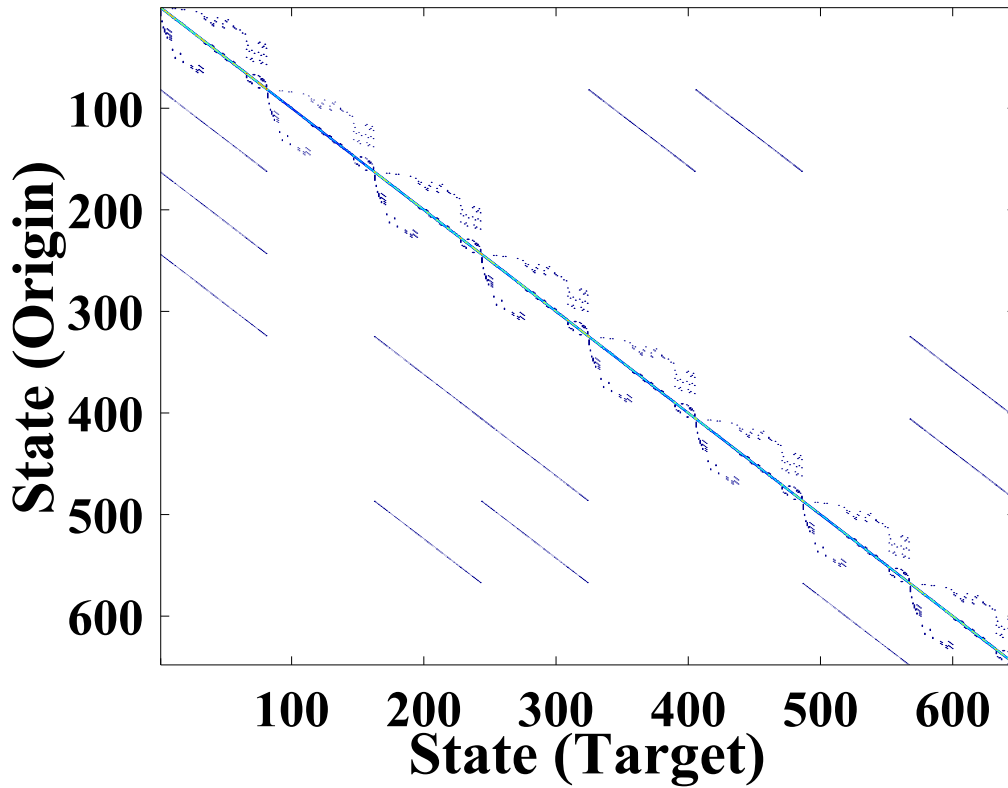
Figure 4.8: Transition matrix contour plot

The computation of LWA proceeds analogously to the prior example. The presentation of the complete DTMC is skipped due to its size. Chapters 5 and 6 present an alternative approach to compute LWA for this example without the necessity to build the full DTMC.

Computing LWA comprises six steps:

1. Build the state space $\mathcal{S} = \{\langle 0, \ldots, 0 \rangle, \ldots, \langle 2, \ldots, 2 \rangle\}$).

2. Compute the transition probabilities between each pair of states to construct the ergodic DTMC $\mathcal{D}$.

3. Compute the stationary distribution $Pr_\Omega(\mathcal{S})$.

4. Specify the desired (safety) predicate $\mathcal{P}$.

5. Prune all transitions departing from legal states (i.e. set them to 0 while their self-targeting transitions are set to 1) to construct $\mathcal{D}_{\text{LWA}}$.

6. Use $Pr_\Omega(\mathcal{S})$ on $\mathcal{D}_{\text{LWA}}$ such that the aggregated probability mass over all legal states after i iterations (i.e. matrix multiplications) computes l_i.
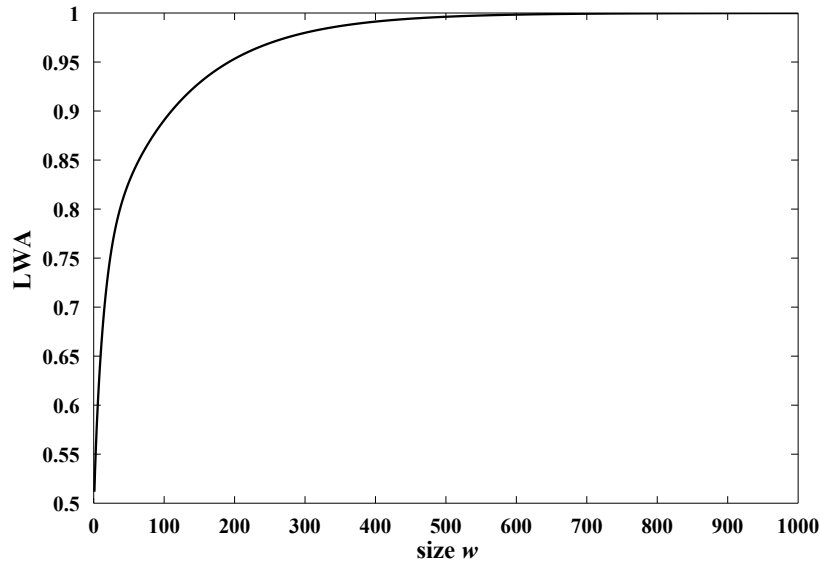
Figure 4.9: Limiting window availability of BASS Example for $w \leq 1000$

Figure 4.9 shows LWA for the first 1000 time window sizes. Figure 4.10 shows the probability mass distribution over time for the illegal states.
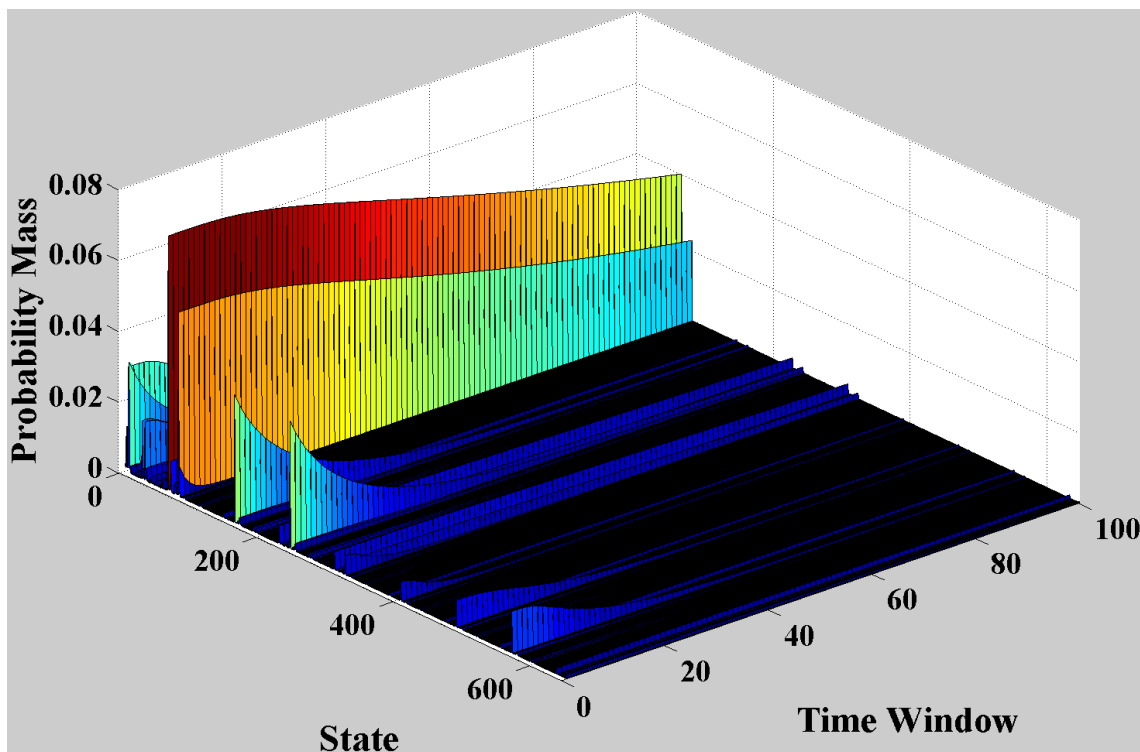

Figure 4.10: Probability mass distribution over time for the illegal states

Constructing $\mathcal{D}_{\text{LWA}}$ vs. reachability

Assume that the initial distribution $Pr_0(\mathcal{S})$ is provided and that the maximal admissible time window w is smaller than the maximal Hamming distance in the system. Then,

the construction of $\mathcal{D}$ is not compulsorily required to its full extent. When the maximal admissible amount of time for reaching a safe state (i.e. w) is smaller than the number of processes, then there are states from which the system deterministically cannot recover *in time*. Such states (and their transitions) can then be omitted.

4.4 Comparing solutions

Assume a variety of different fault-tolerant solutions to the same problem. For instance, a variety of processes with different availabilities can be utilized to build a desired system. The processes have a different cost and different fault probabilities. Different solutions then simply contain all possible permutations of process choices. Alternatively, different solutions can possibly be different structural fault tolerance design compositions.

In order to get the desired degree of fault tolerance at a reasonable price, the trade-offs[10] must be comparable. First, LWA is addressed. Assume two LWA vectors $v_a = \langle l_{a,0}, l_{a,1}, \ldots, l_{a,w} \rangle$ and $v_b = \langle l_{b,0}, l_{b,1}, \ldots, l_{b,w} \rangle$ for two such solutions. The *smaller* function with regards to two LWA vectors is defined as

$$v_a < v_b :\overset{def.}{\Longleftrightarrow} \forall i, 1 \le i \le w : l_{a,i} \le l_{b,i} \wedge v_a \ne v_b \tag{4.12}$$

Proving $v_a < v_b$ can be accomplished even for infinite w via induction.

Let $M_i = (w_i, v_i)$ be a system instance utilizing temporal redundancy w_i and providing the corresponding LWA vector v_i of length w_i. Different system design instances $M_i \ne M_j$ can contain an equal amount of temporal redundancy $w_i = w_j$ and yet carry different vectors v_i. We say that one solution M_i is *strictly* better than another solution M_j if they both contain an equal amount of (temporal) redundancy (i.e. $w_i = w_j$) and M_i has a greater LWA vector:

$$M_i \succ M_j :\Leftrightarrow w_i = w_j \wedge v_i > v_j \tag{4.13}$$

Vice versa, when two solutions M_i and M_j have equal LWA vectors but different amounts of (temporal) redundancy, the solution carrying the smaller amount of redundancy is the *cheaper* option offering an equal amount of fault tolerance (w.r.t. to LWA).

4.5 Summarizing LWA

This section introduced LWA as measure for the efficiency of temporal redundancy. It precisely outlined characteristic properties of LWA such as the stationary distribution being used as initial probability distribution or that it suffices that an execution trace contains one legal state in the desired time window. Three examples, ordered according to their complexity, demonstrated how LWA can be computed. The examples also demonstrated the perils of state space explosion. Finally, in an outlook, the chapter briefly discussed how the LWA vectors of different design solutions can be compared.

---

[10]Each trade-off contains the costs (e.g. redundancy) and fault tolerance (e.g. LWA) of a chosen instance set of system and fault environment.

5. Lumping transition models of non-masking fault tolerant systems

The coverage of computing LWA is inherently confined by the state space explosion, meaning that the size of the Markov chain is exponential in the size of the underlying system, meaning the quantity of processes, their registers and the value domains of the registers.

Lumping, introduced by Kemeny and Snell [Kemeny and Snell, 1976] originally[1] in 1960, is a popular method to cope with the state space explosion. In the context of Markov chains, the concept of lumping is introduced as *probabilistic bisimulation* by Larsen and Skou [Larsen and Skou, 1989] in 1991.

Lumping allows coalescing of *bisimilar* states — which are states that have the same effect on the system — to reduce the size of a DTMC. It facilitates the computation of the relevant measures on the *quotient Markov chain* under lumping equivalence, which is the DTMC in which all bisimilar states have been lumped. This chapter discusses lumping in the context of computing LWA. Parts of it are published [Müllner and Theel, 2011, Müllner et al., 2012, Müllner et al., 2013].

Process and state lumping

Lumping coalesces bisimilar entities and is applicable to both processes in the system model as well as to states in the transition model. This chapter focuses on lumping in the

---

[1]The first edition edition was published in 1960. The second edition from 1976 defines lumpability in Definition 6.3.1 [Kemeny and Snell, 1976, p.124].

transition model. Informally, states in a transition system are *bisimilar* if they have the same effect in the transition model, meaning, they precisely simulate each other's behavior. Formally, states are bisimilar when they have equal transition probabilities regarding their target states and both satisfy and dissatisfy the same predicates.

After this chapter will have prepared lumping of states, chapter 6 determines the relation between *bisimilar processes* — which are processes that "behave equally"[Milner et al., 1992] — in the system model, and *bisimilar states* in the system's transition model. This provides valuable insights to discuss decomposing the system model in the following chapter.

Example

Informally, the information *which* process from a set of bisimilar processes is in a certain state is irrelevant. The information that *one* of them is in a certain condition suffices. For instance, in the BASS example from section 4.3.3, assume two states $s_i = \langle x_1, 2, 0, x_4, x_5, x_6, x_7 \rangle$ and $s_j = \langle x_1, 0, 2, x_4, x_5, x_6, x_7 \rangle$, where the x_i values are pairwise equal. When computing LWA, both states s_i and s_j have an equal effect within the transition model. It is not important to know, *which one* of the process registers R_2 and R_3 is corrupted and which one is not. The information that *one of the process registers is corrupted* and *the other one is not* suffices. States having an equal effect with regards to a specific predicate in a transition model belong to the same *equivalence class* and are *probabilistic bisimilar* (cf. e.g. [Shanks, 1985]). A set of probabilistic bisimilar states can be represented by one state. The process of coalescing bisimilar states is called *lumping* [Kemeny and Snell, 1976]. Lumping states in the transition model allows to reduce the state space. Fortunately, fault-tolerant systems often rely on multiply instantiated homogeneous components that likely offer a great potential for lumping[2].

Related literature

Lumping of probabilistic bisimilar states, based on the definition of probabilistic bisimulation by Larsen and Skou from 1989 [Larsen and Skou, 1989], is presented by Buchholz [Buchholz, 1994] in 1994. Milner introduces lumping in the π-calculus [Milner, 1999] in 1999 in a deterministic setting for processes. Processes qualify for lumping when "they have the same behavior [. . .] for some suitable notion of behavior" (cf. also [Pucella, 2000, Meyer, 2009]).

Lumping and system decomposition are important topics. Popular model checkers like PRISM [Kwiatkowska et al., 2002], CADP [Garavel et al., 2001, Garavel et al., 2011] and MRMC [Katoen et al., 2005] already exploit lumping and decomposition techniques to cope with *large* system and transition models. Katoen et al. [Katoen et al., 2007] provide a general discussion how to generally exploit *bisimulation minimization* — which is minimizing models by exploiting bisimilarity — in the context of applied probabilistic model checking. Computation of window availabilities with probabilistic model checkers has been demonstrated on *instantaneous* window availability exemplarily with PRISM in section 4.1.3.

Yet, instead of feeding system models into a model checker and reasoning about the potential of lumping in general, the goal of this chapter is to conduct the fault tolerance

---

[2]Furthermore, it is important to consider that the relevant predicates do not partition the state space unfavorably as discussed paragraph "Multiple Predicates" on page 70.

analysis *by hand*. This contributes to reason about the relation between fault propagation among processes and bisimilarities in transition models. Furthermore, it shows how lumping can be exploited in the context of non-masking fault tolerant systems in this context to dampen the state space explosion. Discussing the fault tolerance analysis *by hand* allows to understand how computing LWA depends on the system design.

Structure of this chapter

Section 5.1 introduces the notions of *equivalence relation* and *state bisimilarity*. Section 5.2 then applies these notions to discuss lumping in the context of computing the LWA. The most complex part in lumping is the aggregation of transitions. The same section also discusses the *transition lumping* in detail. A small example in section 5.3 demonstrates how lumping can be executed generally. The larger example from section 4.3.3 is continued in the following chapter, including a discussion about system decomposition and the influence of hierarchy on fault propagation. Section 5.4 briefly discusses the benefits and limitations of *approximate* lumping. Section 5.5 concludes this chapter.

5.1 Equivalence classes

Let $\mathcal{D} = \{\mathcal{S}, \mathcal{M}, Pr_0(\mathcal{S})\}$ be a DTMC representing the transition model of a self-stabilizing system as specified in section 4.3.3 with the initial probability distribution being the stationary distribution $Pr_0(\mathcal{S}) = Pr_\Omega(\mathcal{S})$ as discussed in section 4.1. Baier and Katoen define probabilistic bisimilarity as follows:

> A probabilistic bisimulation on $\mathcal{D}$ is an equivalence relation $\mathcal{R}$ on $\mathcal{S}$ such that for all states $s_i, s_j \in \mathcal{R} : L(s_i) = L(s_2) \land pr(\overrightarrow{s_i, T}) = pr(\overrightarrow{s_j, T})$ for each equivalence class $T \in \mathcal{S}/\mathcal{R}$. States s_i, s_j are bisimulation-equivalent (or bisimilar), denoted $s_i \sim_\mathcal{D} s_j$, if there exists a bisimulation $\mathcal{R}$ on $\mathcal{D}$ such that $(s_i, s_j) \in \mathcal{R}$ [Baier and Katoen, 2008, p.808][3].

Here, AP is a set of atomic propositions and $L : \mathcal{S} \to 2^{AP}$ being a labeling function [Baier and Katoen, 2008, p.748]. Two states $s_i, s_j \in \mathcal{S}$ are *bisimilar* with regards to $\mathcal{P}$, if i) both satisfy or both dissatisfy predicate $\mathcal{P}$ and ii) both have equal transition probabilities towards each equivalence class respectively.

Definition 5.1 (State bisimilarity). *Two states s_i and s_j are bisimilar when they satisfy the same predicates and have equal transition probabilities for all transition targets.*

$$\forall s_i, s_j \in \mathcal{S} : s_i \sim s_j :\Leftrightarrow \begin{array}{c} ((s_i \models \mathcal{P} \land s_j \models \mathcal{P}) \lor (s_i \not\models \mathcal{P} \land s_j \not\models \mathcal{P})) \land \\ (\forall d \in \mathcal{S} : \sum_{s \in [d]_\sim} pr(\overrightarrow{s_i, \acute{s}}) = \sum_{s \in [d]_\sim} pr(\overrightarrow{s_j, \acute{s}})) \end{array} \tag{5.1}$$

Bisimilar states can be represented by one state referred to as *lump* (cf. also [Larsen and Skou, 1989, Smith, 2003]). The quotient space, which is the state space in which all bisimilar states are replaced by their respective lumps, is labeled $\mathcal{S}' = \mathcal{S}/\sim$. Figure 5.1 on page 71 provides a small example demonstrating lumping.

---

[3]The symbols have been adapted to suit this thesis.

5.2   Ensuring probabilistic bisimilarity

The idea of the reduction is to construct a Markov chain $\mathcal{D}'$ from $\mathcal{D}$ that is smaller than $\mathcal{D}$ but can also compute the LWA. The reduction method will show that, due to the ergodicity of $\mathcal{M}$, only $\mathcal{M}'$ needs to be computed and that the predicate has to be adapted to fit the new state space.

This section starts by briefly describing *transition lumping*, which is the lumping of the transition matrix. Afterwards, lumping is introduced formally in the context of quantifying fault tolerance measures.

Lumping $\mathcal{D}$

Assume the construction starts with $\mathcal{D}$ and an empty matrix for $\mathcal{D}'$ as inputs. There are three types of transitions to be regarded when computing the transition probabilities for $\mathcal{D}'$:

1. Transitions originating from **non-bisimilar** states targeting **non-bisimilar** states can be transferred to $\mathcal{D}'$ directly. The small circles represent states, the dotted circles equivalence classes and the arrows transitions.

transition in $\mathcal{D}$ **transition in $\mathcal{D}'$**
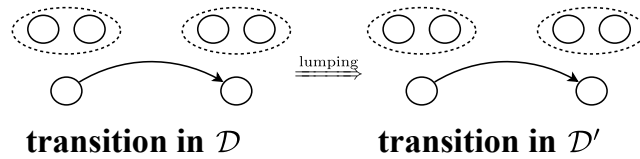
2. Transitions originating from **non-bisimilar** states targeting **states within lumps** now target the lump instead in $\mathcal{D}'$. In case multiple such transitions originate all in one state $s_o : s_o \notin [s]_\sim$ and target multiple states belonging all to the same equivalence class $[s]_\sim$, then their aggregate transition probability becomes the respective sum.

transition in $\mathcal{D}$ **transition in $\mathcal{D}'$**
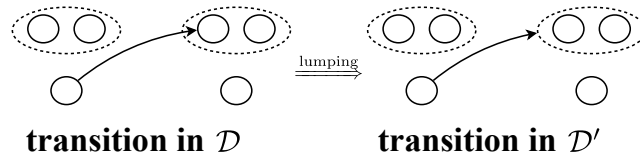
3. Transitions originating from **bisimilar** states are computed as described in equation 5.6.

transition in $\mathcal{D}$ **transition in $\mathcal{D}'$**
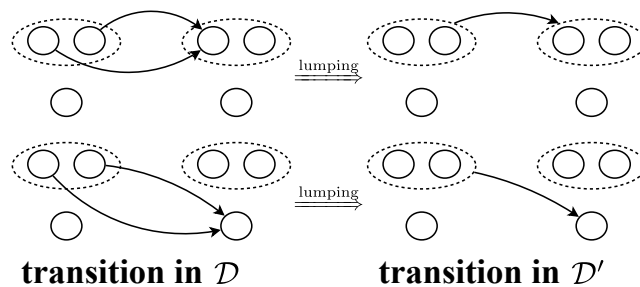
Computing lumped transitions is only required in the third case. All other transitions can be transferred directly from $\mathcal{D}$ to $\mathcal{D}'$. Let $\mathfrak{s}_o = \{s_i, \ldots, s_j\}$ be the origin lump. The target is either an equivalence class $\mathfrak{s}_t = \{s_k, \ldots, s_l\}$, too, or a target state s_t. Each aggregated transition is weighted according to the aggregate weight in the origin states $pr_\Omega([s_i]_\sim) = \sum\limits_{d \in [s_i]_\sim} pr_\Omega(d).$

The formal reduction method

A Markov chain can be lumped and the safety predicate adapted accordingly with the reduction function $red(\mathcal{D}, \mathcal{P})$, as shown in definition 5.2. The set of states from which the aggregated transition *originates* is labeled $[s_o]_\sim$ and the set of *targeted* states is labeled $[s_t]_\sim$.

Definition 5.2 (Reduction). *The reduction comprises seven parts,*

- *the reduction function:*
$$red(\mathcal{D}, \mathcal{P}) = (\mathcal{D}', \mathcal{P}') \tag{5.2}$$
  with $red : \mathcal{S} \to \mathcal{S}'$

- *the reduced DTMC:*
$$\mathcal{D}' = (\mathcal{S}', \mathcal{M}', Pr_0(\mathcal{S}')) \tag{5.3}$$
  with $Pr_0(\mathcal{S}') := Pr_\Omega(\mathcal{S}')$, *and with* $\mathfrak{s}_i, \mathfrak{s}_j \in \mathcal{S}', pr(\overrightarrow{\mathfrak{s}_i, \mathfrak{s}_j}) \in \mathcal{M}' \to [0, 1]$

- *state lumping:*
$$\mathcal{S}' = \{[s]_\sim | s \in \mathcal{S}\} \tag{5.4}$$

- *probability mass lumping:*
$$pr_0([s]_\sim) = \sum_{d \in [s]_\sim} pr_0(d) | \forall s \in \mathcal{S}, \text{ with } pr_0(d) := pr_\Omega(d) \tag{5.5}$$

  with $pr_0([s]_\sim) \to [0, 1]$

- *transition lumping:*
$$pr(\overrightarrow{[s_o]_\sim, [s_t]_\sim}) = \frac{\sum\limits_{d_i \in [s_o]_\sim} \sum\limits_{d_j \in [s_t]_\sim} pr(\overrightarrow{d_i, d_j}) \cdot pr(d_i)}{\sum\limits_{d_i \in [s_o]_\sim} pr(d_i)} \tag{5.6}$$

  with $pr(\overrightarrow{[s_o]_\sim, [s_t]_\sim}) \to [0, 1]$

- *predicate lumping:*
$$[s]_\sim \models \mathcal{P}' :\Leftrightarrow \exists d \in [s]_\sim : d \models \mathcal{P} \tag{5.7}$$

The reduction $red(\mathcal{D}, \mathcal{P})$ shown in definition 5.2 in equation 5.2 reduces the DTMC $\mathcal{D}$ and adapts the predicate $\mathcal{P}$ accordingly. The reduced DTMC $\mathcal{D}'$ consists of a reduced state space $\mathcal{S}'$ and correspondingly adapted transitions — both regarded in the three following equations — as shown in equation 5.3. An initial probability distribution as in definition 2.7 is not required here as the resolving transition system is an ergodic Markov chain

that applies its stationary as initial distribution. Equation 5.4 describes the *state lumping* (including the aggregation of the probability masses w.r.t. the equivalence classes shown in equation 5.5). The initial probability distribution over $\mathcal{S}$ is aggregated, such that the probability mass of all states within each equivalence class is summed up to compute the initial probability mass for the lumped states in $\mathcal{S}'$ as shown in equation 5.5. Those states belonging to the same equivalence class $[s]_\sim$ are aggregated and their transitions are computed respectively as shown in equation 5.6. Equation 5.6 describes the *transition lumping*. It simply states that the probability of any equivalence class C to class D is $pr(\overrightarrow{s, D}) = \sum\limits_{s' \in D} pr(\overrightarrow{s, s'})$. Equation 5.7 is only provided for sake of completeness[4] and follows directly from definition 5.1.

The weight terms can be canceled based on the law of total probability (LTP, cf. e.g. [Pfeiffer, 1978, p.47]) for conditional probabilities[5] and with the conditions for states to be bisimilar (cf. definition 5.1):

$$pr(\overrightarrow{[s_o]_\sim, [s_t]_\sim}) = \frac{\sum\limits_{d_i \in [s_o]_\sim} \sum\limits_{d_j \in [s_t]_\sim} pr(\overrightarrow{d_i, d_j}) \cdot pr(d_i)}{\sum\limits_{d_i \in [s_o]_\sim} pr(d_i)} \stackrel{\text{Def. 5.1}}{\Longrightarrow} \sum\limits_{d_i \in [s_o]_\sim} pr(\overrightarrow{d_i, d_j}) \qquad (5.8)$$

Section 5.3 provides a demonstrative and simple example explaining why the weighting terms can be canceled.

Lumping preserves the ability to compute the LWA

In the model checking community it is common knowledge that a quotient transition system under an equivalence relation preserves desired attributes with regards to the equivalence relation [Baier and Katoen, 2008, p.459]. This section shows that the ergodic *quotient* transition model *of a non-masking fault tolerant system* preserves the desired attributes, which in this case implies preservation of the ability to compute LWA.

Informally, the proof shows that both $\mathcal{D}$ and $\mathcal{D}'$ progress equally over time with bisimilar initial distributions and with respect to the equivalence relation. When progress is equal in each time step, they have bisimilar stationary distributions, too. Finally, with bisimilar distributions and bisimilar progress, both $\mathcal{D}$ and $\mathcal{D}'$ compute the same LWA. The first assumption is that the order in which i) computing the LWA and ii) lumping are executed does not matter, or else the reduction would not be bisimilar.

Theorem 5.1 (Commutativity of lumping and calculating the stationary distribution).
Computing the stationary distribution with subsequent lumping leads to the same result as first lumping and then computing the stationary distribution.

$$\forall [s]_\sim \in \mathcal{S}' : pr_\Omega([s]_\sim) = \sum\limits_{d \in [s]_\sim} pr_\Omega(d) \qquad (5.9)$$

---

[4]With the conditions specified in definition 5.1, the $\exists$ quantifier in equation 5.7 can be replaced with an $\forall$ quantifier (i.e. if one state of the equivalence class satisfies the predicate, then all states must satisfy the predicate).

[5]Soudjani and Abate [Soudjani and Abate, 2013b, eq.7] exploit the same opportunity in a similar context.

The $pr_k(s_i)$ function, which is the probability mass in state s_i at time t, is overloaded by allowing a set of states as input referring to the aggregated probability mass within the set of states.

Theorem 5.1 is implicitly verified in proof 5.1 by showing that both the original and the reduced Markov chain have an *equal stationary probability distribution* — with regards to their particular equivalence classes — by induction. The proof is twofold. It first shows that for any initial probability distribution both DTMCs show bisimilar progress and therefore converge to *bisimilar stationary distributions* (i.e. $Pr_\Omega(\mathcal{S}) \stackrel{\text{eq. 5.5}}{\Longrightarrow} Pr_\Omega([s]_\sim)$). Then, given that both DTMCs have a bisimilar stationary distribution and provide bisimilar progress, it is simple to show that the quotient Markov chain preserved the ability to compute the LWA.

For any initial probability distribution $Pr_0(\mathcal{S})$, the corresponding initial probability distribution for $\mathcal{S}'$ is computed with equation 5.5. This provides the anchor for the proof. The induction step shows that the lumped transitions *do the same* as the original transitions, meaning that they cause bisimilar progress.

Proof 5.1 (Equivalence of stationary distributions)**.**
Let $Pr_0(\mathcal{S})$ be an arbitrary initial distribution for $\mathcal{D}$ and let $Pr_0([s]_\sim) = \sum\limits_{d \in [s]_\sim} pr_0(d)$ be an initial distribution for $\mathcal{D}'$. Show that for $Pr_k(\mathcal{S})$ and $Pr_k([s]_\sim)$ — which are the probability distributions for $\mathcal{D}$ and $\mathcal{D}'$ at time step k — the following holds:

$$\forall k \geq 0, \forall [s]_\sim \in \mathcal{S}' : pr_k([s]_\sim) = \sum_{d \in [s]_\sim} pr_k(d) \tag{5.10}$$

Proof per induction over k.

Anchor: $k = 0$ holds by assumption (cf. equation 5.5).

Step: show that the following holds

  Assumption:

$$pr_{k+1}([s]_\sim) = \sum_{[d]_\sim \in \mathcal{S}'} pr_k([d]_\sim) \cdot pr(\overrightarrow{[d]_\sim, [s]_\sim}) \tag{5.11}$$

$$= \sum_{[d]_\sim \in \mathcal{S}'} \left(\sum_{e \in [d]_\sim} pr_k(e) \right) \cdot \left(\sum_{f \in [s]_\sim} pr(\overrightarrow{d, f}) \right) \tag{5.12}$$

$$= \sum_{[d]_\sim \in \mathcal{S}'} \sum_{e \in [d]_\sim} \sum_{f \in [s]_\sim} pr_k(e) \cdot pr(\overrightarrow{d, f}) \tag{5.13}$$

and with $pr(\overrightarrow{e, f}) = pr(\overrightarrow{d, f})$ (with e and d being bisimilar, cf. definition 5.1)

$$= \sum_{[d]_\sim \in \mathcal{S}'} \sum_{e \in [d]_\sim} \sum_{f \in [s]_\sim} pr_k(e) \cdot pr(\overrightarrow{e, f}) \tag{5.14}$$

$$= \sum_{e \in \mathcal{S}} \sum_{f \in [s]_\sim} pr_k(e) \cdot pr(\overrightarrow{e, f}) \tag{5.15}$$

$$= \sum_{f \in [s]_\sim} \sum_{e \in \mathcal{S}} pr_k(e) \cdot pr(\overrightarrow{e, f}) \tag{5.16}$$

$$= \sum_{d \in [s]_\sim} pr_{k+1}(d) \tag{5.17}$$

Thereby, $\forall k \geq 0, \forall [s]_\sim \in \mathcal{S}' : pr_k([s]_\sim) = \sum\limits_{d \in [s]_\sim} pr_k(d)$. The corresponding equality for the stationary distributions follows. $\qquad\square$

The anchor of the proof holds by equation 5.5 in definition 5.2. The lumped states simply aggregate the probability mass that the states they comprise contain. The step shows that both the original and the reduced system converge to the same probability distribution with regards to the equivalence relation. Thereby, both DTMCs also have a probabilistic bisimilar stationary distribution.

Corollary 5.1 (Equivalence of l_0)**.**
Theorem 5.1 and the two conditions of definition 5.1 imply that the limiting availability l_0 satisfies $l_0(\mathcal{D}, \mathcal{P}) = l_0(\mathcal{D}', \mathcal{P}')$ (with respect to the equivalence classes). Thereby, $l_0(\mathcal{D}, \mathcal{P}) = \sum\limits_{s \models \mathcal{P}} pr_\Omega(s)$ and consequently $l_0(\mathcal{D}', \mathcal{P}') = \sum\limits_{[s]_\sim \models \mathcal{P}'} pr_\Omega([s]_\sim)$.

The final step is to show that both $\mathcal{D}$ and $\mathcal{D}'$ compute the same LWA. The proof exploits the previous proof that showed both DTMCs have bisimilar progress.

Theorem 5.2 (Equivalence of LWA)**.**
For each $n \in \mathbb{N}$, $l_n(\mathcal{D}, \mathcal{P}) = l_n(\mathcal{D}', \mathcal{P}')$.

Proof 5.2 (Equivalence of LWA)**.**
The proof follows immediately from definition 4.1 (LWA) plus theorem 5.1, applied to the stationary distributions $Pr_\Omega(\mathcal{S})$ and $Pr_\Omega(\mathcal{S}')$ as initial distributions. $\qquad\square$

Therefore, both the original DTMC $\mathcal{D}_{\text{LWA}}$ and the lumped DTMC $\mathcal{D}'_{\text{LWA}}$ compute the same LWA. With lumping, information that is not relevant for computing the LWA is abstracted. The reduction is an irreversible surjective function regarding the conditions specified in definition 5.1, resulting in a probabilistic bisimilar quotient DTMC $\mathcal{D}'$.

Each state in the domain $\mathcal{S}$ is mapped to a state in the co-domain $\mathcal{S}'$ but not vice versa. The reduction step is irreversible. The full product chain $\mathcal{D}$ cannot be created from $\mathcal{D}'$.

Multiple predicates

It might be desirable to evaluate more than one predicate at a time, like for instance operability and safety as discussed in paragraph "Predicating desired properties" on page 54. Such systems are also known as *mixed criticality systems* (cf. e.g. [Baruah et al., 2012]). States belong to the same equivalence class, if they satisfy or dissatisfy each predicate uniformly:

Definition 5.3 (Mixed criticality bisimulation)**.**

$$\forall s_i, s_j \in \mathcal{S} : s_i \sim s_j :\Leftrightarrow \begin{aligned} &(\forall \mathcal{P}_k : ((s_i \models \mathcal{P}_k \wedge s_j \models \mathcal{P}_k) \vee (s_i \not\models \mathcal{P}_k \wedge s_j \not\models \mathcal{P}_k)) \wedge \\ &\quad (\forall d \in \mathcal{S} : \sum\limits_{s \in [d]_\sim} pr(\overrightarrow{s_i, s}) = \sum\limits_{s \in [d]_\sim} pr(\overrightarrow{s_j, s})) \end{aligned}$$

Each predicate partitions the state space. Mixed criticality systems are further discussed in the future work section in chapter 8. Analyzing systems according to multiple predicates does not increase the complexity of the analysis, assuming that the predicate partitioning of the state space is not influencing with lumping and decomposition. Therefore, this thesis continues with one predicate.

The double-stroke alphabet

This paragraph introduces an alternative labeling of lumped states that increases the readability in the upcoming examples. Instead of labeling a lumped state with the equivalence class it constitutes, lumped states are labeled with identifiers from the double-stroke alphabet (e.g. $\mathbb{1}, \mathbb{2}, \mathbb{3}, \ldots$) or with abstract identifiers (e.g. $\mathbb{s}_i, \mathbb{s}_j$) to refer to coerced register partitions. For instance, let $s_i = \langle 0, 2, 0 \rangle$ and $s_j = \langle 0, 0, 2 \rangle$ be two bisimilar states $s_i \sim s_j$. Then, the corresponding lump is labeled $\mathbb{s}_i = \langle 0, \mathbb{2} \rangle$. A *coerced register partition* is that set of registers within bisimilar states in which the states differ. In the example, that partition contains the second and third registers. In the examples provided in this thesis, a double-stroke integer refers to the sum over the values stored in each such register partition. The labeling is valuable for the examples in this thesis but might be ambiguous for others. A counterexample is provided in appendix A.5.5.

5.3 Example

Lumping is discussed in more detail on a large example in the following chapter in connection with system decomposition. This section demonstrates lumping on a small example with two isomorphic — and thereby bisimilar — states. A similar example in a different context is used in [Graf et al., 1996, p.8]. Consider the DTMC shown in figure 5.1(a) as the transitional model of a two process system. The safety predicate for this example demands both registers to store 0.

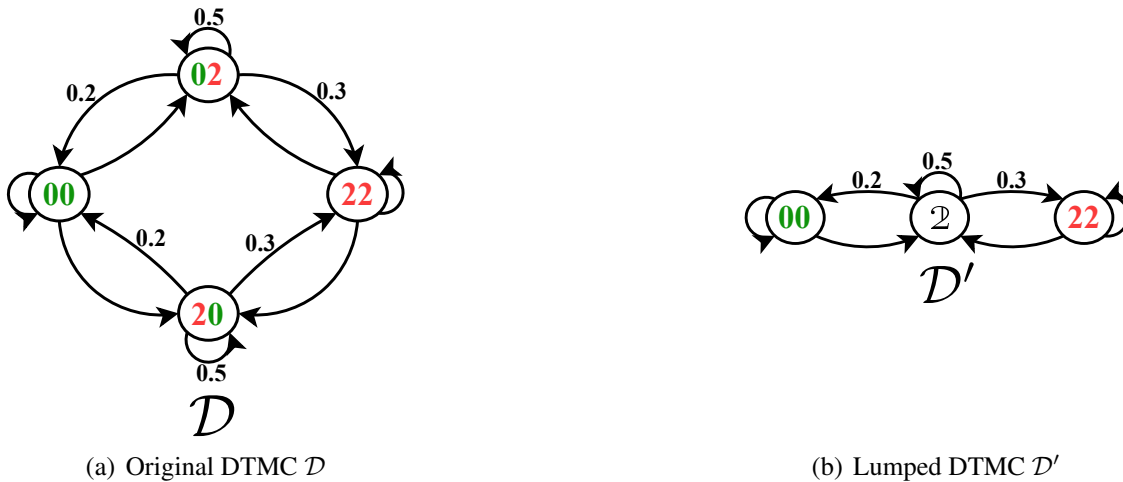(a) Original DTMC $\mathcal{D}$ (b) Lumped DTMC $\mathcal{D}'$

Figure 5.1: Small lumping example

In this example, the states $\langle 0, 2 \rangle$ and $\langle 2, 0 \rangle$ can be lumped. Irrelevant transition probabilities are not shown to increase the readability. According to definition 5.1, states $\langle 2, 0 \rangle$ and $\langle 0, 2 \rangle$ are probabilistic bisimilar if and only if:

- their transition probabilities regarding each target state are equal and

- they both satisfy or both dissatisfy $\mathcal{P}$.

The transitions originating from $\langle 2, 0 \rangle$ and $\langle 0, 2 \rangle$ are respectively equal and both states dissatisfy $\mathcal{P}$. The states are probabilistic bisimilar and are replaced by one state that

is labeled $\overline{2}$. For demonstration we assume different weights: $pr_\Omega(\langle 0, 2 \rangle) = 0.3$ and $pr_\Omega(\langle 2, 0 \rangle) = 0.4$. All transitions *targeting* one of the lumpable states target the lump instead without further adaptation. All transitions that originate in the lumpable states originate from the lump now and are computed with equation 5.6. The following three transitions compute $p(\overrightarrow{\langle \overline{2} \rangle, \langle 0, 0 \rangle})$, $p(\overrightarrow{\langle \overline{2} \rangle, \langle \overline{2} \rangle})$, and $p(\overrightarrow{\langle \overline{2} \rangle, \langle 2, 2 \rangle})$:

$$p(\overrightarrow{\langle \overline{2} \rangle, \langle 0, 0 \rangle}) = \frac{0.2 \cdot 0.3 + 0.2 \cdot 0.4}{0.3 + 0.4} = \frac{0.2 \cdot (0.3 + 0.4)}{0.3 + 0.4} = 0.2 \qquad (5.18)$$

$$p(\overrightarrow{\langle \overline{2} \rangle, \langle \overline{2} \rangle}) = \frac{0.5 \cdot 0.3 + 0.5 \cdot 0.4}{0.3 + 0.4} = \frac{0.5 \cdot (0.3 + 0.4)}{0.3 + 0.4} = 0.5 \qquad (5.19)$$

$$p(\overrightarrow{\langle \overline{2} \rangle, \langle 2, 2 \rangle}) = \frac{0.3 \cdot 0.3 + 0.3 \cdot 0.4}{0.3 + 0.4} = \frac{0.3 \cdot (0.3 + 0.4)}{0.3 + 0.4} = 0.3 \qquad (5.20)$$

The weight of the new lumped state is the aggregated weight of the states that constitute the lumped state, which is in this case:

$$pr_\Omega(\langle \overline{2} \rangle) = pr_\Omega(\langle 0, 2 \rangle) + pr_\Omega(\langle 2, 0 \rangle) = 0.7 \qquad (5.21)$$

The example demonstrates that the *weights* — which is the aggregated probability mass of the states of a lump according to the stationary distribution — of the lumpable states are canceled.

Reachability vs. equivalence class identification

In the above example $\langle 2, 2 \rangle$ was not considered to be part of the lump from the beginning. Considering serial execution semantics, it cannot belong to $\overline{2}$ according to its Hamming distance which is different from the one of $\langle 0, 2 \rangle$ and $\langle 2, 0 \rangle$. The benefit of serial execution semantics is that the identification of equivalence classes can be focused to *reachability classes* regarding the legal set of states (considering multiple predicates partitioning the state space, cf. the corresponding paragraph on page 70). While states belonging to equivalence class $\overline{2}$ can converge to the legal state in *one* computation step, state $\langle 2, 2 \rangle$ requires at least *two* steps. Clustering the states according to their Hamming distance, described in paragraph "Hamming Distance" on page 15, to the closest legal state simplifies the search for equivalence classes to the relevant clusters. Furthermore, when i) the initial probability distribution is already known and ii) the maximal admissible window size is smaller than the maximal Hamming distance towards the legal set of states, then the DTMC requires to be constructed only as far from the legal states with descending Hamming distance as the admissible time window reaches. Benoit et al. [Benoit et al., 2006] propose a similar technique in 2006 for continuous time Markov chains under the compositional construction with the Kronecker product. This thesis requires a different approach for i) employing DTMCs and ii) as the Kronecker product is not generally applicable in the context of this thesis as explained later in paragraph "Serial DTMC composition operator $\otimes$" on page 87. The idea, nevertheless, is the same.

5.4   Approximate bisimilarity

When states are not precisely bisimilar but only (sufficiently) similar, and lumping is required in order to reduce a system to become tractable for analysis, *approximate* lumping

might be a good choice. While Jou and Smolka [Jou and Smolka, 1990] introduce approximate bisimulation generally in 1990, Girard and Pappas [Girard and Pappas, 2005] add a discussion for linear systems in 2005 and D'Innocenzo et al. [D'Innocenzo et al., 2012] discuss approximate lumping with regards to PCTL. When similar states are lumped *pessimistically* regarding the predicates, the measures that are then computed with the lumped transition system are a conservative approximation. In some cases it is possible to prove that desired properties are satisfied *at least* within certain boundaries, although the system is likely perform better regarding the properties. As previously discussed, fault-tolerant systems typically contain homogeneously redundant components and thus, also likely provide a large potential for precise lumping. Hence, this thesis focuses on probabilistic bisimilarity.

5.5 Summarizing lumping

This chapter presented equivalence classes and probabilistic bisimulation for the transition models of non-masking fault tolerant systems. It was shown that lumping preserves the ability to compute the LWA and the possibility to tackle multiple predicates simultaneously was discussed. The double-stroke alphabet for lumped states was introduced and a small example demonstrated lumping. Next to lumping, limiting the construction of the DTMC to the reachable partition of the maximal admissible time window was presented as an opportunity to simplify the analysis. Finally, the opportunity of approximate lumping was briefly discussed.

Although lumping is a well known technique that has been generally introduced in model checking, it is important to have discussed lumping in the context of this thesis. Now, the following chapter can exploit the results of this chapter to discuss the decomposition of *hierarchically* depending systems, like self-stabilizing systems, to show how lumping can be applied locally on the transition models of subsystems. This will be one of the most interesting points in the following chapter. While lumping and compositional construction Markov models is often presented for systems with mutually *independent* processes, this thesis aims at *hierarchically* ordered systems.

6. Decomposing hierarchical systems

In order to avoid the computation of lumping equivalences on intractably large Markov chains, a system decomposition is proposed that splits the system into subsystems. The goal is to exploit lumping *locally* on the considerably smaller Markov chains of the subsystems. Recomposing the lumped Markov chains of the subsystems constructs a lumped transition model of the whole system, thus avoiding the construction of the full product chain. Parts of this chapter are published [Müllner and Theel, 2011, Müllner et al., 2012, Müllner et al., 2013].

Section 2.4 introduced the construction of a DTMC $\mathcal{D}$ modeling the behavior of a deterministic system dynamics under probabilistic influence as shown in figure 6.2.

$$\text{fault model, scheduler}$$

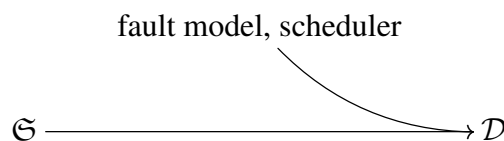$$\mathfrak{S} \longrightarrow \mathcal{D}$$

Figure 6.1: DTMC construction, section 2.4

After chapter 3 discussed fault tolerance and the scope of this thesis to motivate LWA as suitable measure in this context, chapter 4 showed how to compute LWA by adapting the Markov chain as shown in figure 6.2:

fault model, scheduler

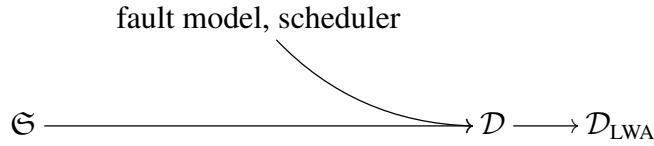$$\mathfrak{S} \longrightarrow \mathcal{D} \longrightarrow \mathcal{D}_{\mathrm{LWA}}$$

Figure 6.2: Computing LWA without lumping, chapter 4

Chapter 5 discussed the reduction of the Markov chain and showed that the ability to precisely compute LWA is preserved as shown in figure 6.3:
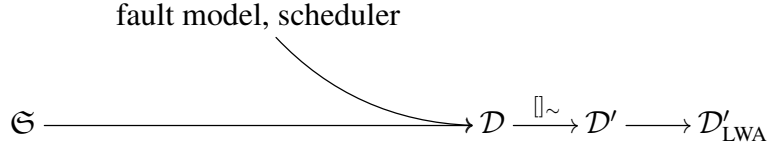
fault model, scheduler

$$\mathfrak{S} \longrightarrow \mathcal{D} \xrightarrow{\;[\![]\!]\sim\;} \mathcal{D}' \longrightarrow \mathcal{D}'_{\mathrm{LWA}}$$

Figure 6.3: Lumping, chapter 5

Yet, either $\mathcal{D}$ can be constructed — which means LWA can be computed and lumping is not necessary — or the construction of $\mathcal{D}$ is intractable. In that case, sampling methods as presented in paragraph "Sample-based analysis via simulation" on page 50 are a reasonable option to acquire results.

When sample-based methods are insufficient as precise results are required from an analysis, then it is desirable to find a solution to make lumping applicable without the necessity to construct the full Markov chain. This chapter proposes a general approach to system decomposition with the intention to compute LWA. It introduces the decomposition function $\tau_k(\mathfrak{S})$, which takes a system topology as input and provides a set of *overlapping subsystems* as output. Then, the Markov chains for all subsystems are constructed. From here on they are referred to as *sub-Markov chains*[1]. The sub-Markov chains are considerably smaller than the full product chain $\mathcal{D}$, making lumping far more likely to be applicable. The intention of the decomposition is that it is *lossless*. Contrary to lumping, no information is abstracted from the transition model. Recomposing the subsystems or sub-Markov chains should result in the original system or full product chain. Thereby, bisimilarity with regards to safety predicate $\mathcal{P}$ is preserved.

The *overlap of subsystems* is important to account for the fault propagation as well as for the convergence among the subsystems. Furthermore, a re-composition function $\otimes(\{\mathcal{D}_1, \ldots\})$ is presented. It is a matrix multiplication similar to the Kronecker product which accounts for the underlying execution semantics. In that sense, the Kronecker product provides a *parallel composition* while the $\otimes$ function provides a *hierarchical composition* for serial execution semantics.

When lumping is not applied on the sub-Markov chains, then recomposing them with the $\otimes$ function yields the full product chain $\mathcal{D}$ of the previously decomposed system as shown in figure 6.4. The system topology is sliced according to some function $\tau_k(\mathfrak{S})$ into overlapping subsystems that are given as subsets of processes of Π. Instead of writing $\tau_k(\mathfrak{S}) = \{\{\Pi_1, E_1, \mathfrak{A}_1\}, \ldots\}$ we write $\tau_k(\mathfrak{S}) = \{\{\Pi_1, \Pi_2, \ldots\}, E, \mathfrak{A}\}$ for short.

---

[1] In related literature, these are also referred to as *marginals*.

fault model, scheduler

$$\mathfrak{S} \xrightarrow{\tau_k(\mathfrak{S})} \{\{\Pi_1, \ldots, \Pi_n\}, E, \mathfrak{A}\} \longrightarrow \{\mathcal{D}_1, \ldots \mathcal{D}_n\} \xrightarrow{\otimes} \mathcal{D} \longrightarrow \mathcal{D}_{\mathrm{LWA}}$$
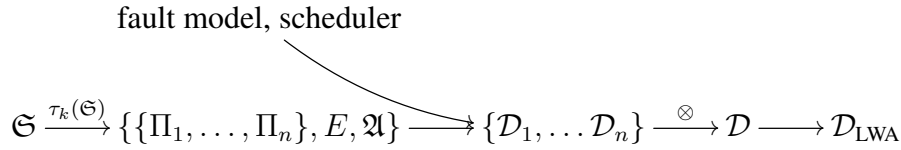
Figure 6.4: Lossless system decomposition and transition model re-composition

Both approaches from figures 6.2 and 6.4 have the same inputs and outputs, showing that the intention behind system decomposition and sub-Markov chain re-composition is to be lossless.

The final stage is to apply lumping on the sub-Markov chains before the full product chain is constructed as shown in figure 6.5. Since decomposition is lossless and therefore bisimilar, and lumping is also bisimilar, the whole procedure is bisimilar, too. Lumping is a congruence with regards to the proposed composition. When $\mathcal{D}_1$ is bisimilar to $\mathcal{D}'_1$, then the composed DTMCs $\mathcal{D} = \mathcal{D}_1 \otimes \mathcal{D}_2$ and $\mathcal{D}' = \mathcal{D}'_1 \otimes \mathcal{D}_2$ are bisimilar as well (cf. [Buchholz, 1997][2]).

fault model, scheduler

$$\mathfrak{S} \xrightarrow{\tau_k(\mathfrak{S})} \{\{\Pi_1, \ldots, \Pi_n\}, E, \mathfrak{A}\} \longrightarrow \{\mathcal{D}_1, \ldots \mathcal{D}_n\} \xrightarrow{\llbracket \rrbracket_\sim} \{\mathcal{D}'_1, \ldots \mathcal{D}'_n\} \xrightarrow{\otimes} \mathcal{D}' \longrightarrow \mathcal{D}'_{\mathrm{LWA}}$$
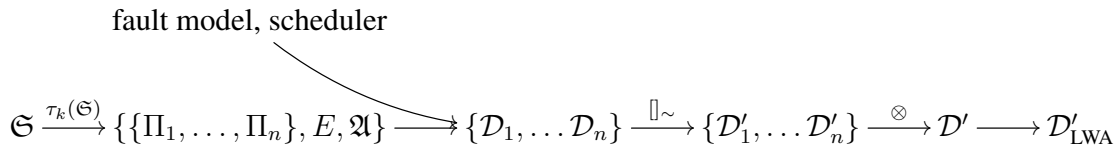
Figure 6.5: Combining decomposition and lumping

At first glance, the decomposition seems to increase the complexity of computing the LWA by adding further steps. Figure 6.2 contained only two steps — construction of the Markov chain and adapting it to compute the LWA — while figure 6.5 comprises five steps, which are

1. decomposition,

2. construction of the sub-Markov chains,

3. local lumping,

4. re-composition of the lumped sub-Markov chains and

5. adapting it to compute the LWA.

Yet, these additional steps allow to circumvent the necessity to construct the full product Markov chain. After discussing the general approach, this chapter provides a figurative example in section 6.5 demonstrating on the BASS example from section 4.3.3 how the complexity of computing LWA can be drastically decreased.

---

[2]Buchholz discusses the congruence for Petri nets.

Related work

One of the earliest approaches to minimize the transition models of subsystems was proposed by Graf et al. [Graf et al., 1996] in 1996. Their work on "minimisation" of finite state distributed systems "produces processes that are smaller in the specification implementation preorder" [Graf et al., 1996, p.24]. They demonstrate the "minimisation" on a small example of two processes exchanging messages via a buffer. "Minimisation" allows to lump the states of the two buffers, resulting in one shared buffer for both processes. Their setting of two communicating processes is similar to the TLA setting in section 2.5. The decomposition as it is proposed in this thesis differs in two major points: First, this thesis provides examples for systems comprising more than two processes. Extending their example seems equally complex to extending the TLA discussed in section 2.5. As discussed there, *two* mutually depending processes are a special case of heterarchy that is not trivially extended. Second, this thesis provides a more distinguished composition operator that does not necessarily rely on a parallel composition.

One important system characteristic regarding the *decomposability* is how the processes depend on each other. When processes are mutually *independent*, system decomposition is *arbitrary*. Every process can be represented by its own DTMC and since the DTMCs of the processes do not influence each other, they can be composed and lumped arbitrarily. Their composition is *parallel*. On the other hand, when processes — like in some self-stabilizing systems — depend on each other, fault propagation and convergence make the sub-Markov chains depend on each other as the processes — and therefore the subsystems — are not independent. The subsystems and their corresponding sub-Markov chains are ordered hierarchically and cannot be composed in parallel. Their composition must be carried out *hierarchically*.

Hermanns and Katoen [Hermanns and Katoen, 1999] provide a compositional approach in 1999/2000 for analyzing *independent* processes, in this case "a plain-old telephone system" [Hermanns and Katoen, 1999, p.14] which is similar to the work by Erlang [Erlang, 1909, Erlang, 1917]. Their approach shows very well how drastically and simply the size of a transition model can be reduced with *independent* processes. While certain "interactions can only appear when all participants are ready to engage in it" [Hermanns and Katoen, 1999, p.10], meaning that participants synchronize on some actions, the basic system functionality of participants does not rely on synchronization. Section 7.1 provides a case study in a similar context.

In 2002, Garavel and Hermanns [Garavel and Hermanns, 2002] utilize the Caesar/Aldebaran Development Package (CADP) [Garavel et al., 2001, Garavel et al., 2011] to carry out formal verification and performance analysis with the non-stochastic process algebra LOTOS [198, 1989], extended by a few additional operators (one of them being "minimisation" [Garavel and Hermanns, 2002, p.20]), and combined with the tool BCG_MIN for reducing the transition model. They demonstrate the application in the context of "the SCSI-2 bus arbitration protocol" [Garavel and Hermanns, 2002, ch.3], in which seven disks share one bus with a controller. This work shows the practical value of the tools and the importance of reasoning about decomposition strategies to analyze desired system properties. Similar to previous work [Hermanns and Katoen, 1999], the property of process independence is exploited for parallel composition.

Benoit et al. [Benoit et al., 2006] discuss limited reachability, similar to the discussion in paragraph "Reachability of states" on page 13, to cope with large product chains. Contrary

to the analysis in this thesis, they also focus on a parallel composition with the Kronecker product.

Boudali et al. provide publications [Boudali et al., 2007a, Boudali et al., 2007b, Boudali et al., 2008a, Boudali et al., 2008b, Boudali et al., 2009, Boudali et al., 2010] between 2007 and 2010 focusing on a modular approach to evaluate the dependability of systems based on the CADP background. Their work on using "dynamic fault trees" to construct "input/output interactive Markov chains" (IO-IMCs) [Boudali et al., 2007b] provides for a modular analysis of systems to avoid "vulnerability to state-space explosion" and facilitates a modular model construction. They extend their work by case studies in [Boudali et al., 2007a]. One key aspect is:

> "*Compositional modeling* (4a) entails that a model can be created by composing smaller sub-models. There are two important types of composition: parallel composition, which combines two or more components which are at the same level of abstraction, and hierarchical composition, where one component is internally realized as a combination of sub-components." [Boudali et al., 2008a, p.244]

In this publication from 2008 [Boudali et al., 2008a] they introduce the *Arcade* formalism (ARChitecturAl Dependability Evaluation) as an extension to their previous work to discuss "the requirements that a suitable formalism for dependability modeling/evaluation should posses. [...] The *Arcade* modeling language incorporates both parallel and hierarchical composition." Although the authors claim that the "hierarchical composition will be realized" and point out that "aggressive aggregation (also called lumping or bisimulation minimization)" is important in this context, the *hierarchical* composition is not discussed. In [Boudali et al., 2008b], a *sequential* composition of transition models of subsystems is proposed, exploiting lumping after each *parallel* composition [Boudali et al., 2008b, ch.4]. This sequential composition is facilitated by a *composer tool* which uses CADP. In 2009, they analyze [Boudali et al., 2009] the availability of distributed software systems, "where the software designer simply inputs the software module's decomposition annotated with failure and repair rates." The repair of components here does not rely on the functionality of other components. Similarly, their later work [Boudali et al., 2010] also discusses only *parallel* composition [Boudali et al., 2010, sec.3.2]. Although this set of publications discusses important topics such as

- motivating IO-IMCs as suitable transition model for analyzing nonfunctional properties like fault tolerance,

- the exploitation of popular tools such as CADP, and

- pointing out that minimization is crucial in the analysis,

it does solely focus on parallel composition.

This thesis focuses on evaluating the recovery within a system of mutually depending processes based on DTMCs, as motivated in paragraphs "Restricting communication via guards" on page 7 and "Execution Semantics" on page 11. Fault tolerance is not always achieved *locally*, but can also be achieved via dependability among components of a system. This latter case is intrinsically a harder problem which is addressed in this chapter.

DTMCs are the selected transition model in this thesis. While the work by Boudali et al. is based on IO-IMCs, Rakow [Rakow, 2011] discusses coping with the state space explosion on Petri nets. The two techniques she proposes are *Petri-net slicing* and *cutvertex reduction*. Their work shows the importance of selecting *the right* transition model, as discussed in the future work section in chapter 8.

Contributions and limitations

The core contributions of this chapter are

- classifying systems according to their fault propagation as either

    1. hierarchical, meaning with unidirectional fault propagation,

    2. semi-hierarchical, meaning with neither uni- nor fully omnidirectional fault propagation,

    3. heterarchical, meaning omnidirectional fault propagation, or

    4. independent, meaning with no fault propagation,

- reasoning about general decomposition guidelines for hierarchically structured systems,

- showing that the proposed decomposition is lossless, thus preserving the ability to compute LWA, and

- discussing the main variants of semi-hierarchical systems and how they still can possibly benefit from the system decomposition.

This chapter provides general guidelines for system decomposition. It does not show how tools like PRISM and CADP have to be adapted to cope with hierarchic composition or discuss how *optimal* slicing can be achieved.

Structure of this chapter

Section 6.1 discusses the hierarchy among processes introduced by self-stabilization. Section 6.2 extends the notation by the terms *subsystem* and *sub-Markov chain* and introducing their relation. Section 6.3 discusses general guidelines for the decomposition of system models. Section 6.4 shows that the decomposition process preserves probabilistic bisimilarity as presented in figure 6.4. The BASS example from section 4.3.3 is used to exemplify decomposition, local lumping and re-composition. Section 6.5 demonstrates how the example system is decomposed, lumping is locally applied on the sub-Markov chains of the subsystems and the fault propagation from *superior* subsystems into *inferior* subsystems is accounted for. The example allows to figuratively address and explain the challenges that are discussed before. Section 6.6 reasons about the connection between hierarchical fault propagation and decomposability. With hierarchical systems being fully decomposable on the one side and heterarchical systems being not at all decomposable on the other side, semi-hierarchical systems possibly provide some exploitable leverage to cope with otherwise intractable systems. Section 6.7 concludes this chapter.

6.1 Hierarchy in self-stabilizing systems

Detection and correction of the effects of faults are means of fault tolerance as discussed in section 3.1. The fault masker introduced in section 3.5 covers for detection. Fault tolerance design is like constructing an equation. The right hand side of the equation, the *purchased commodity*, is the degree of faults that the system can cope with maskingly, or the degree to which faults are corrected in time. Figuratively, the *price tag* on a fault tolerance design, the left hand side of the equation, shows two currencies: temporal and spatial redundancy. Chapter 3 refined the focus to temporal redundancy. Probabilistic self-stabilization, as introduced in section 3.2, provides a possible formal foundation to discuss the relation between temporal redundancy and degree of masking fault tolerance.

The classification of self-stabilizing systems according to fault propagation is proposed in [Müllner et al., 2012, Müllner et al., 2013]. It is extended in this thesis by distinguishing dependent from independent processes. In systems through which the effects of faults propagate, processes rely on each other. For instance, the TLA assigns a value to the executing process always with regards to the register of the other process. Similar to the scenario by Graf et al. [Graf et al., 1996], here all processes rely on even *every* other processes, which for two processes is trivial. Another case of depending processes was presented with the BASS example, in which processes relied on each other not fully meshed, but hierarchically. In case the algorithm executed by the processes of the system does not consult foreign registers to derive the local value, the processes are considered to be *independent*. Notably, mixed mode systems are possible. Sub-systems with dependencies are further distinguished into three classes according to the fault propagation within the (sub-)system: i) hierarchical shown in figure 6.6(a), ii) semi-hierarchical shown in figure 6.6(b) and iii) heterarchical, either directly shown in figure 6.6(c) or indirectly shown in figure 6.6(d).

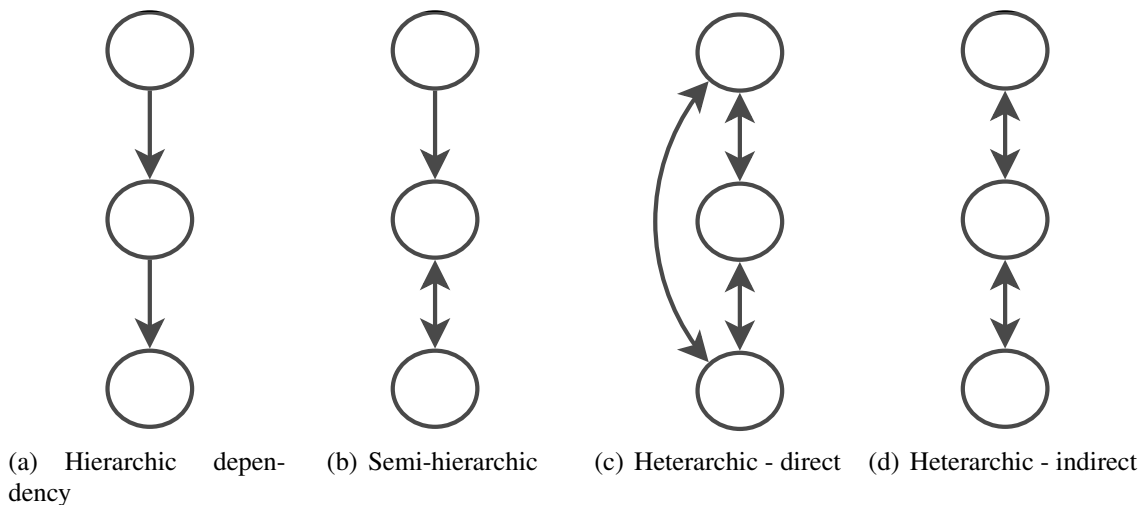(a) Hierarchic dependency   (b) Semi-hierarchic   (c) Heterarchic - direct   (d) Heterarchic - indirect

Figure 6.6: Different dependency types

- In a **hierarchical** system, the processes are topologically ordered. For instance, the examples presented in this thesis, except the TLA, feature a designated *root* process which does not rely on any other process and is the only *independent* process. The processes are semi-ordered according to their distance to this root. Each non-root process only accepts information from processes that are closer to the root than

itself. Therefore, the effects of faults strictly propagate from the *root* towards the *leaf* processes, which are the processes with the greatest minimal distance to the root on their branch. The decomposition starts in the subsystem containing the root process and sequentially progresses towards those subsystems containing the leaf processes. While this chapter focuses on systems with one root process for sake of clarity, an extension to systems with multiple root processes in the future work section in chapter 8.

- In **heterarchical** systems, all processes are *peers* and influence each other either directly or indirectly. Every process provides information to every other process, possibly via processes. The effects of faults propagate *omni-directionally*. Since every process relies on every other process, decomposition is highly complex.

- **Semi-hierarchical** systems are not globally heterarchical, but are neither globally hierarchical. One example are hierarchically structured heterarchical *sub*systems. Section 6.6.1 introduces further notions of semi-hierarchy and generally discusses the possibilities for decomposing them.

The benefits of hierarchical fault propagation can be exploited to allow for a combination of lumping and decomposition. Hierarchical fault propagation transforms a system topology into a directed acyclic graph (DAG) in which processes communicate only according to the hierarchy as discussed in paragraph "Restricting communication via guards" on page 7. When decomposing such a system, the property of unidirectional fault propagation becomes an invaluable asset.

While hierarchy is common among self-stabilizing systems — generally being enforced via unique identifications (e.g. BASS, cf. algorithm 4.6) — heterarchical systems are uncommon when processes are supposed to cooperate in order to provide for fault tolerance. One heterarchical example that is presented in this thesis, though, is the TLA, introduced in section 2.5. Its task is to establish a hierarchy — which here means alternating access to the crossing — among an otherwise heterarchical system.

Motivating slicing with overlapping processes

Slicing becomes necessary when the full transition model is intractable. With the subsystems being hierarchically depending, faults propagate only in one direction from root towards leafs. Decomposing the hierarchical system by slicing allows to treat the overlapping processes as *gateways of fault propagation*, channeling the effects of faults that processes closer to the root have on processes that are farther away from the root. From the perspective of a leaf process it does not matter *why* its *superior* process acts the way it does — meaning how it is influenced by other processes and fault propagation — but only *how* it acts.

Assume a process in a hierarchical system that is neither root nor leaf, referred to as *transient* process hereafter. Slicing the system in that process allows to first compute how the process behaves as a leaf process of the superior subsystem. Then, all influence from unidirectional fault propagation of the superior processes is accounted for. After that, the process can act as the *root* process for the inferior processes.

6.2 Extended notation

This section

- formally introduces the terms *subsystem*, *residual process* and *overlapping process* in paragraph "Subsystems, residuals and sets of overlapping process",

- relates subsystems according to fault propagation and their respective position in the system in paragraph "Inferior and superior positions", and

- further distinguishes subsystems into *root*, *transient* and *leaf* subsystems in paragraph "Root, transient and leaf subsystems".

Paragraph "Overlapping sets" elaborates on processes overlapping. This issue is important for reasoning about decomposition strategies in section 6.3. But before that, this section further extends the formal notation by a *decomposition* function $\tau_k(\mathfrak{S})$ introduced in paragraph "Decomposition function $\tau_k(\mathfrak{S})$" and a decomposition function $\otimes$ (colloquially *o-times*) proposed in paragraph "Serial DTMC composition operator $\otimes$".

Subsystems, residuals and sets of overlapping process

A system $\mathfrak{S}$ comprises processes $\Pi = \{\pi_1, \ldots, \pi_n\}$ and is sliced into subsystems $\tau_k(\mathfrak{S}) = \{\{\Pi_1, \ldots, \Pi_m\}, E, \mathfrak{A}\}$. The decomposition operator $\tau_k(\mathfrak{S})$ is explained in detail in paragraph "Decomposition function $\tau_k(\mathfrak{S})$". The system is sliced into subsets of processes Π_i (referred to as subsystem from hereon). This means that between each two processes of a subsystem there exists a path via edges such that every process is reachable[3]. Subsystems share at least one process with another subsystem in which they overlap. Each subsystem contains at least one process that is not shared with any other subsystem. Processes belonging exclusively to one subsystem are referred to as *residual* process (or just *residual* for short). Processes belonging to multiple subsystems are referred to as *overlapping* processes.

The term *slicing* in this context means to divide the set of processes into partially overlapping subsystems. Contrary, the term *uncoupling* refers to the extraction of the sub-Markov chains accounting for overlapping processes from their corresponding sub-Markov chains as explained later in paragraph "Uncoupling with $\otimes$" on page 89.

The following figure 6.7 depicts the terms that are introduced in this paragraph. Subsystems containing besides residuals only overlapping processes, such that no overlapping process has read access to a register from a process not belonging to its own subset, are referred to as *root subsystems*. They commonly contain a root process and are not influenced by other subsystems. Subsystems containing besides residuals only overlapping processes from which no process outside the subsystem reads, are referred to as *leaf subsystems*. These are commonly *farthest away* from the root subsystem. The effects of faults are propagated into them from the whole system, yet the effects of faults do not emanate from them back into the system (i.e. other adjacent subsystems). All other subsystems are referred to as *transient subsystems*.

---

[3]The distinction between the system model and the transition model is important. The processes of a subsystem in the system model must be a connected component, cf. also [Baier and Katoen, 2008, p.96].

System $\mathfrak{S}$

root subsystem
superior

transient subsystem
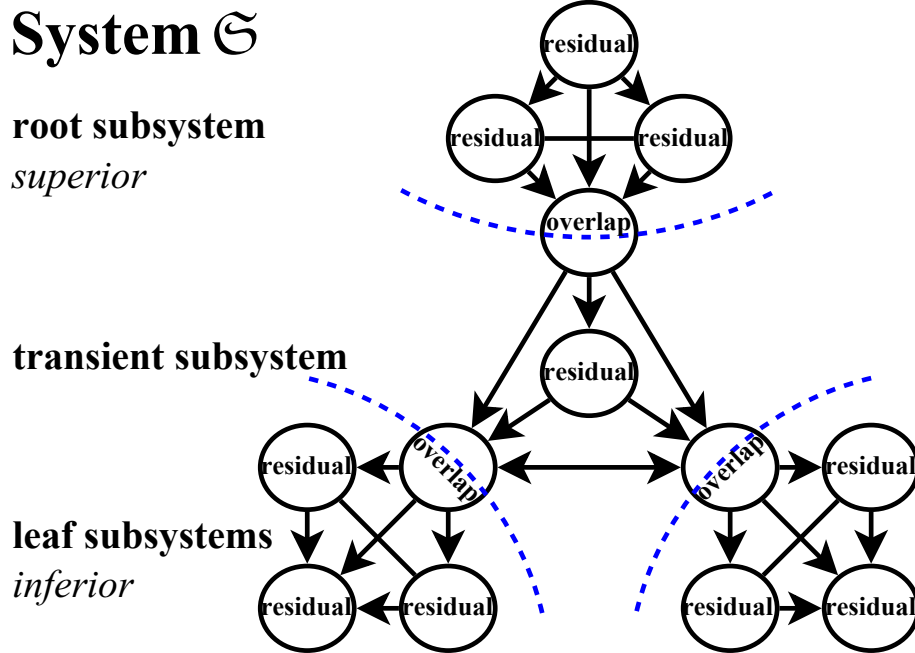
leaf subsystems
inferior

Figure 6.7: Extended notation - example

Figure 6.7 provides an illustrative example to explain the new terms. The dotted (blue) lines indicate the aspired slicing. The root subsystem contains four processes depicted as circles on the top. The top three of them are the residuals and the remaining process is an overlapping process. The effects of faults propagate from the root through the transient into the leaf subsystems. Figuratively, overlapping processes are the gateways (or more precisely: valves for the case of unidirectional fault propagation) of fault propagation. General guidelines for system decomposition are discussed in section 6.3.

Inferior and superior positions

A subsystem or process that can propagate faults (directly or indirectly) into other subsystems or processes is *superior* to these subsystems or processes. The subsystems or processes that are prone to possible fault propagation from superior subsystems or (overlapping) processes are *inferior* to these. When subsystems or processes possibly propagate faults mutually into one another, they must commonly not be decomposed. Constructing independent transition models from interrelated processes is not in the focus of this thesis. Exceptions are discussed in section 6.6.

Root, transient and leaf subsystems

Some self-stabilizing systems work with a fixed hierarchy — for instance via unique identifiers — while others, like self-stabilizing leader election (LE) [Fischer and Jiang, 2006, Nesterenko and Tixeuil, 2011], switch the leader role among the processes from time to time. When the role of the root process can be switched among the processes, the direction of fault propagation changes accordingly. Newton's cradle is an illustrative example to address the latter case as shown in figure 6.8. When the role of the root is switched to another process the direction of fault propagation changes accordingly.
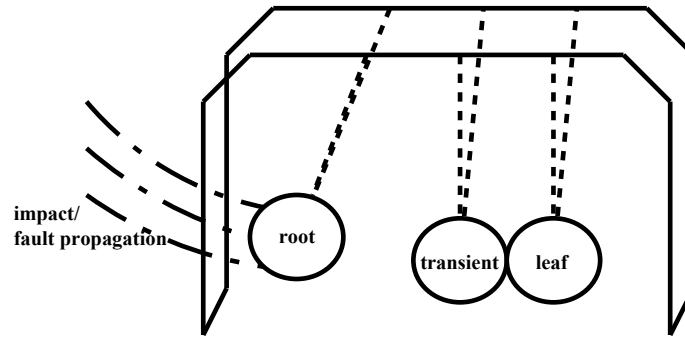
Figure 6.8: Newton's cradle and fault propagation

Overlapping sets

Consider the system topology presented in figure 6.9 executing the BASS. The superior root subsystem Π_1 propagates the effects of faults into the inferior leaf subsystems Π_2 to Π_7 via the sets of processes in which the subsystems overlap $\{\pi_5\}$, $\{\pi_6\}$, $\{\pi_7, \pi_8\}$ and $\{\pi_9, \pi_{10}\}$.
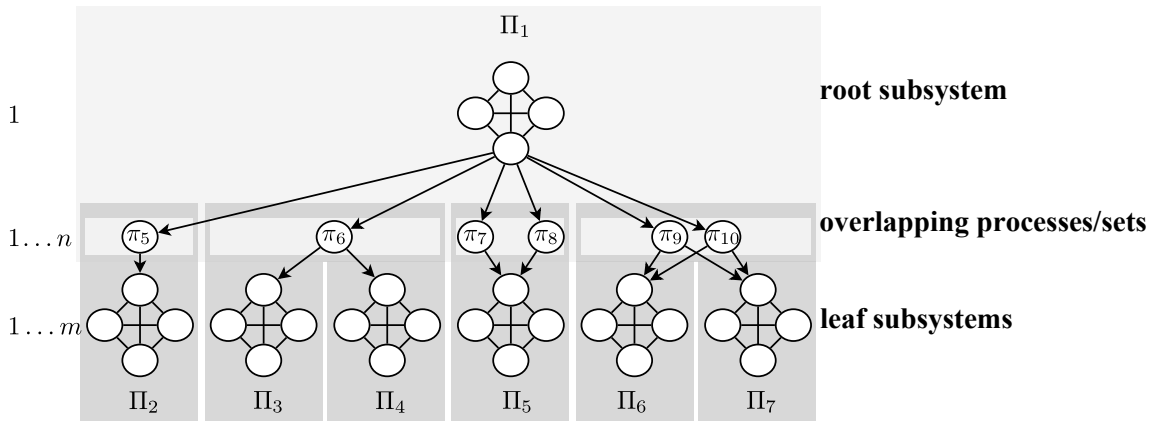

Figure 6.9: Classifying decomposition possibilities via overlapping sets

The superior subsystem Π_1 propagates into inferior subsystems either through one process (e.g. π_5 or π_6) or through multiple processes (e.g. π_7 and π_8, or π_9 and π_{10}) into either one subsystem (e.g. Π_2 or Π_5) or multiple subsystems (Π_3 and Π_4, or Π_6 and Π_7). The case where multiple superior subsystems *channel* their fault propagation through one or more overlapping processes does not occur. The *overlapping* process would depend on both superior subsystems. Being mutually dependent, the joint set of superior subsystems and overlapping process would be regarded as not decomposable in the context of this thesis. The possible relation cardinalities are of the form[4] $\langle$superior, overlapping, inferior$\rangle = \langle 1, 1 \ldots n, 1 \ldots m \rangle$. A set of overlapping processes is referred to as an *overlapping set* when it contains all *relevant*[5] overlapping processes that two subsystems, one superior and one inferior subsystem, share. Notably, multiple overlapping sets might overlap with one another. For instance, if π_8 and π_9 were together replaced by one process,

---

[4]The relation cardinality notation stems from the entity relationship model [Chen, 1976].

[5]*Relevant* here means in the context of two overlapping subsystems all processes that are part of both subsystems.

then both rightmost overlapping sets would overlap in that process. This case is discussed in detail in paragraph "Non-overlapping sets of overlapping processes" on page 91 and depicted in figure 6.11. Overlapping processes belong to all their subsystems initially, for instance $\pi_5 \in \Pi_1 \wedge \pi_5 \in \Pi_2$. When the sub-Markov chains have been constructed, the influence of each overlapping process is to be awarded to one sub-Markov chain exclusively as explained in paragraph "Uncoupling with $\otimes$" on page 89, or otherwise it would be multiply accounted for.

Decomposition function $\tau_k(\mathfrak{S})$

Commonly, there are multiple options to decompose a system, or else, the system would be sufficiently small to analyze it without the need for decomposing it. Let the set of all *applicable* decompositions be $\tau(\mathfrak{S}) = \{\tau_1(\mathfrak{S}), \ldots\}$. A decomposition $\tau_k(\mathfrak{S})$ is applicable when it slices the system into at least one root subsystem and one leaf subsystem and all contain at least one residual and overlap with at least one other subsystem. From $\tau(\mathfrak{S})$, one *distinct* applicable decomposition rule $\tau_k(\mathfrak{S})$ is selected. The conditions for a decomposition rule to be applicable or even to be optimal are discussed in section 6.3. The selected decomposition slices the set of processes Π of a system $\mathfrak{S}$ into sets of subsystems $\tau_k(\mathfrak{S}) = \{\{\Pi_1, \ldots\}, E, \mathfrak{A}\}$. As discussed at the beginning of this chapter we write $\{\{\Pi_1, \ldots\}, E, \mathfrak{A}\}$ as short form of $\{\{\Pi_1, E_1, \mathfrak{A}_1 \ldots\}\}$. Furthermore, the set of processes Π as sole input does usually not sufficient. Some schedulers or algorithms might exclude certain decompositions. Therefore, scheduling must implicitly be regarded during the decomposition.

Notably, not all self-stabilizing systems necessarily qualify for decomposition. A self-stabilizing counter example class are ring topologies. They are, for instance, a precondition to the self-stabilizing variant of the mutual exclusion algorithm (MutEx) [Pnueli and Zuck, 1986, Brown et al., 1989, Arora and Nesterenko, 2004, Lamport, 1986c]. Due to cyclic dependencies (i.e. there are no superior processes), the cyclic topology cannot be sliced. Similarly, heterarchical systems cannot be sliced, too, due to mutual dependencies.

Bernstein conditions

This paragraph briefly presents Bernstein conditions [Bernstein, 1966], introduced by Bernstein in 1966, in the light of this thesis. It discusses, how they can be exploited in reasoning about decomposability, and their limitations.

In the area of parallel computing, the Bernstein conditions specify if two *program segments* are independent and can be executed in parallel [Bernstein, 1966]. The basic idea is that two program segments P_i and P_j with inputs I_i and I_j and outputs O_i and O_j are *independent* when:

$$I_j \cap O_i = \emptyset \tag{6.1}$$

$$I_i \cap O_j = \emptyset \tag{6.2}$$

$$O_i \cap O_j = \emptyset \tag{6.3}$$

In case no Bernstein condition is violated, the subsystems are independent and can be analyzed individually and composed arbitrarily as discussed in paragraph "Related work"

on page 78. In case only Bernstein condition 6.1 is violated, which is introduced by Bernstein as *flow dependency*, the system can be decomposed. It coincides with this thesis' notion of *unidirectional fault propagation*. Analogously, condition 6.2, which is called *anti-dependency*, formalizes the opposed. The dependency between the two subsystems is simply switched and a hierarchic decomposition is applicable. In case condition is 6.3 violated, which is introduced by Bernstein as *output dependency*, a decomposition is impossible with the methods discussed in this thesis. Notably, *programs* might temporarily switch between satisfying and dissatisfying Bernstein conditions.

Serial DTMC composition operator $\otimes$

The composition of sub-Markov chains is intricate. Liu and Trenkler [Liu and Trenkler, 2008] provide a survey discussing various matrix products. One of them, the Kronecker product, is applicable here for the special case of *maximal parallel execution semantics*. For instance, assume the following two transition matrices:

$$\mathcal{M}_1 = \begin{pmatrix} 0.2 & 0.8 \\ 0.7 & 0.3 \end{pmatrix}, \text{ and } \mathcal{M}_2 = \begin{pmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{pmatrix} \tag{6.4}$$
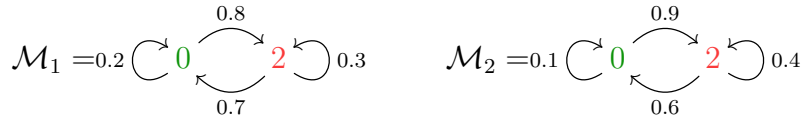
Figure 6.10: DTMC construction, section 2.4

Both matrices model processes that are independent of one another, executing BASS for instance. Despite their independence, they execute *simultaneously*, meaning that each time step every enabled process executes. In this particular case, a parallel composition is suitable. Let $\otimes_K$ denote the Kronecker product only for this example. Then

$$\mathcal{M}_1 \otimes_K \mathcal{M}_2 = \begin{pmatrix} 0.2 \cdot \begin{pmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{pmatrix} & 0.8 \cdot \begin{pmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{pmatrix} \\ 0.7 \cdot \begin{pmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{pmatrix} & 0.3 \cdot \begin{pmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0.02 & 0.18 & 0.08 & 0.72 \\ 0.12 & 0.08 & 0.48 & 0.32 \\ 0.07 & 0.63 & 0.03 & 0.27 \\ 0.42 & 0.28 & 0.18 & 0.12 \end{pmatrix}$$

$$\tag{6.5}$$

The Kronecker product suits maximal parallel execution semantics as every process can change its register together with other processes, which is also known as synchronous composition. Yet, if the Kronecker product is applied under lesser parallel execution semantics, it results in an aggregate matrix containing positive transition probabilities for transitions between states with a greater Hamming distance than permissible via the execution semantics. This issue is demonstrated comprehensively in the example in section 6.5.

To account for serial execution semantics, the composition operator is required to prevent such transitions. In this thesis, an adapted variant of the Kronecker product labeled $\otimes$ is introduced, accruing to the Kronecker product under maximal parallel execution semantics. The $\otimes$ operator is presented in algorithm 16 in MatLab notation.

Let $|\mathcal{S}| = |\mathcal{S}_1| \cdot |\mathcal{S}_2|$ be the size of the aggregate state space and $\mathfrak{s}_{\Pi_i}$ be the probability for a process within subsystem Π_i to be selected. The transition matrices $\mathcal{M}_1$ and $\mathcal{M}_2$ of the corresponding sub-Markov chains $\mathcal{D}_1$ and $\mathcal{D}_2$ are iterated row by row and column by column, thereby resulting in four for-loops.

Algorithm 6.1 (The $\otimes$ operator)**.**

```
1  M = zeros(|S|);
2  for j = 1 : |S₁| do
3      for l = 1 : |S₂| do
4          for i = 1 : |S₁| do
5              for k = 1 : |S₂| do
6                  if i ≠ j ∧ l ≠ k then
7                      M((j-1)·|S₂|+l, (i-1)·|S₂|+l) =
8                          M((j-1)·|S₂|+l, (i-1)·|S₂|+l)+
9                          M₁(j,i)·M₂(l,k)·𝔰_Π₁;
10                     M((j-1)·|S₂|+l, (j-1)·|S₂|+k) =
11                         M((j-1)·|S₂|+l, (j-1)·|S₂|+k)+
12                         M₁(j,i)·M₂(l,k)·𝔰_Π₂;
13                 else
14                     M((j-1)·|S₂|+l, (i-1)·|S₂|+k) =
15                         M((j-1)·|S₂|+l, (i-1)·|S₂|+k)+
16                         M₁(j,i)·M₂(l,k);
```

The first line initializes an empty matrix $\mathcal{M}$ in the dimensions of the product matrix $|\mathcal{S}|$. Then, the transition models of the sub-Markov chains $\mathcal{D}_1$ and $\mathcal{D}_2$ are iterated row by row and column by column. Each permutation of cells within $\mathcal{M}_1(i,j)$ and $\mathcal{M}_2(k,l)$ is computed as specified in algorithm 16. The cases in which more than one register changes are distinguished, regarding the relative scheduler selection probabilities, into those cases of exclusively either register changing. This is accounted for in the code in lines 7 to 12.

We return to the previous example that explained the Kronecker product. This time, we compute $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$. The green filled cells correspond to the first assignment in the if block in lines 7 to 9. The yellow filled cells correspond to the second assignment in the if block in lines 10 to 12. The red filled cells correspond to the third assignment in the else block in lines 14 to 16.

i	j	k	l				
if				line 7	line 8	line 9	
				line 10	line 11	line 12	
else				line 14	line 15	line 16	
1	1	1	1	$\mathcal{M}(1,1)=$	$\mathcal{M}(1,1)+$	$\mathcal{M}_1(1,1)\cdot$	$\mathcal{M}_2(1,1)$
1	1	2	1	$\mathcal{M}(1,2)=$	$\mathcal{M}(1,2)+$	$\mathcal{M}_1(1,1)\cdot$	$\mathcal{M}_2(1,2)$
2	1	1	1	$\mathcal{M}(1,3)=$	$\mathcal{M}(1,3)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(1,1)$
2	1	2	1	$\mathcal{M}(1,3)=$	$\mathcal{M}(1,3)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(1,2)$ $\cdot\mathfrak{s}_{\Pi_1}$
				$\mathcal{M}(1,2)=$	$\mathcal{M}(1,2)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(1,2)$ $\cdot\mathfrak{s}_{\Pi_2}$
1	1	1	2	$\mathcal{M}(2,1)=$	$\mathcal{M}(2,1)+$	$\mathcal{M}_1(1,1)\cdot$	$\mathcal{M}_2(2,1)$
1	1	2	2	$\mathcal{M}(2,2)=$	$\mathcal{M}(2,2)+$	$\mathcal{M}_1(1,1)\cdot$	$\mathcal{M}_2(2,2)$
2	1	1	2	$\mathcal{M}(2,4)=$	$\mathcal{M}(2,4)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(2,1)$ $\cdot\mathfrak{s}_{\Pi_1}$
				$\mathcal{M}(2,1)=$	$\mathcal{M}(2,1)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(2,1)$ $\cdot\mathfrak{s}_{\Pi_2}$
2	1	2	2	$\mathcal{M}(2,4)=$	$\mathcal{M}(2,4)+$	$\mathcal{M}_1(1,2)\cdot$	$\mathcal{M}_2(2,2)$
1	2	1	1	$\mathcal{M}(3,1)=$	$\mathcal{M}(3,1)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(1,1)$
1	2	2	1	$\mathcal{M}(3,1)=$	$\mathcal{M}(3,1)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(1,2)$ $\cdot\mathfrak{s}_{\Pi_1}$
				$\mathcal{M}(3,4)=$	$\mathcal{M}(3,4)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(1,2)$ $\cdot\mathfrak{s}_{\Pi_2}$
2	2	1	1	$\mathcal{M}(3,3)=$	$\mathcal{M}(3,3)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(1,1)$
2	2	2	1	$\mathcal{M}(3,4)=$	$\mathcal{M}(3,4)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(1,2)$
1	2	1	2	$\mathcal{M}(4,2)=$	$\mathcal{M}(4,2)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(2,1)$ $\cdot\mathfrak{s}_{\Pi_1}$
				$\mathcal{M}(4,3)=$	$\mathcal{M}(4,3)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(2,1)$ $\cdot\mathfrak{s}_{\Pi_2}$
1	2	2	2	$\mathcal{M}(4,2)=$	$\mathcal{M}(4,2)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(2,2)$
2	2	1	2	$\mathcal{M}(4,3)=$	$\mathcal{M}(4,3)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(2,1)$
2	2	2	2	$\mathcal{M}(4,4)=$	$\mathcal{M}(4,4)+$	$\mathcal{M}_1(2,1)\cdot$	$\mathcal{M}_2(2,2)$

Table 6.1: Computing the aggregated matrix

Table 6.1 shows how the execution semantics sensitive product is computed in equation 6.6.

$$
\mathcal{M} = \begin{pmatrix}
0.2 \cdot 0.1 & 0.2 \cdot 0.9 + 0.8 \cdot 0.9 \cdot \mathfrak{s}_{\Pi_2} & 0.8 \cdot 0.1 + 0.8 \cdot 0.9 \cdot \mathfrak{s}_{\Pi_1} & 0 \\
0.2 \cdot 0.6 + 0.8 \cdot 0.6 \cdot \mathfrak{s}_{\Pi_2} & 0.2 \cdot 0.4 & 0 & 0.8 \cdot 0.6 \cdot \mathfrak{s}_{\Pi_1} + 0.8 \cdot 0.4 \\
0.7 \cdot 0.1 + 0.7 \cdot 0.9 \cdot \mathfrak{s}_{\Pi_1} & 0 & 0.3 \cdot 0.1 & 0.3 \cdot 0.9 + 0.7 \cdot 0.9 \cdot \mathfrak{s}_{\Pi_2} \\
0 & 0.7 \cdot 0.6 \cdot \mathfrak{s}_{\Pi_1} + 0.7 \cdot 0.4 & 0.7 \cdot 0.6 \cdot \mathfrak{s}_{\Pi_2} + 0.3 \cdot 0.6 & 0.3 \cdot 0.4
\end{pmatrix}
\tag{6.6}
$$

Assuming that the scheduler selects both processes with the same probability, $\mathcal{M}$ is computed as shown in equation 6.7

$$
\mathcal{M} = \begin{pmatrix}
0.02 & 0.54 & 0.44 & 0 \\
0.35 & 0.08 & 0 & 0.56 \\
0.385 & 0 & 0.03 & 0.585 \\
0 & 0.49 & 0.39 & 0.12
\end{pmatrix}
\tag{6.7}
$$

The comparison between both product chains from equation 6.5 and 6.7 shows that the aggregate transition probability is evenly divided among the probabilities that exclusively either of the processes executes according to the evenly distributed scheduling probability.

The labeling of sub-Markov chains

The final addition to the notation regards the labeling of sub-Markov chains. Each set of overlapping processes is to be awarded to one sub-Markov chain exclusively or else it would by accounted for too often. Therefore, it must be *uncoupled* (i.e. extracted) from all sub-Markov chains except one. With unidirectional fault propagation, it is convenient to uncouple sets of overlapping processes from superior subsystems and award them to one inferior subsystem exclusively.

A sub-Markov chain carrying all its sets of overlapping processes is simply labeled $\mathcal{D}_i$, referring to its set of processes Π_i. Uncoupled sub-Markov chains, which are the sub-Markov chains of sets of overlapping processes, are labeled with the set of processes they refer to, for instance $\mathcal{D}_{\pi_j}$ for single processes or $\mathcal{D}_{\{\pi_j, \pi_j\}}$ for sets with more than one process. Those sub-Markov chains from which the influence from all sets of overlapping processes haven been uncoupled are labeled with a minus sign, for instance $\mathcal{D}_{i,-}$. Consider the example in figure 6.9. First, $\mathcal{D}_1$ is constructed including all sets of overlapping processes. Then, the influence form the particular overlapping sets is uncoupled, leaving $\mathcal{D}_{1,-}$, $\mathcal{D}_{\pi_5}$, $\mathcal{D}_{\pi_6}$, $\mathcal{D}_{\{\pi_7, \pi_8\}}$ and $\mathcal{D}_{\{\pi_9, \pi_{10}\}}$. Then, the sub-Markov chains of the inferior subsystems are computed with the sub-Markov chains of the sets of overlapping processes. The sub-Markov chains of *shared* overlapping sets of processes, which are $\mathcal{D}_{\pi_6}$ and $\mathcal{D}_{\pi_9, \pi_{10}}$ in this case, finally have to be awarded exclusively to one of the inferior subsystems. For instance, π_6 can be awarded to Π_3 resulting in $\mathcal{D}_3$ and $\mathcal{D}_{4,-}$ and π_9 and π_{10} can be awarded to Π_6 resulting in $\mathcal{D}_6$ and $\mathcal{D}_{7,-}$ (or vice versa). An example in section 6.5 shows how the labeling is applied.

Uncoupling with $\otimes$

As discussed in the previous paragraph, overlapping processes are to be awarded to one subsystem exclusively. This process is referred to as *uncoupling*. Despite its application to (re-)composing sub-Markov chains, the $\otimes$ operator is further employed in the uncoupling. For instance, when the system shown in figure 6.9 is decomposed with τ_k and $\mathcal{D}_1$ has been

constructed from Π_1 and the relevant information[6], the sets of overlapping processes have to be uncoupled from $\mathcal{D}_1$ before computing the leaf sub-Markov chains. Assume a DTMC $\mathcal{D}_i$ to account for a set of processes Π_i. The goal of *uncoupling* is to arrive at multiple sub-Markov chains that each account for a subset of processes — or more generally process registers — exclusively.

Let $\mathcal{M}$ be a DTMC from which $\mathcal{M}_1$ is uncoupled. Uncoupling is a surjective function that lumps those states, in which the values stored by the processes of the *uncoupled* states coincide. In the context of this thesis — that is by labeling states via process registers — $\mathcal{M}_1$ is uncoupled from $\mathcal{M}$ by lumping all states in which the registers are equal. Equation 5.6 computes the aggregated transition probabilities. For instance, consider the BASS example from section 4.3.3. Uncoupling a DTMC containing the first six processes would exemplarily lump $\langle 0,0,0,0,0,0,0\rangle$, $\langle 0,0,0,0,0,0,1\rangle$ and $\langle 0,0,0,0,0,0,2\rangle$ in $\mathcal{M}$ to form $\langle 0,0,0,0,0,0\rangle$, and analogously for all other states which coincide in the first six digits.

Figuratively, consider an observer to watch the process registers. The processes change their register from one value to another value with a certain probability. The corresponding Markov chain contains all these probabilities in its transition probability matrix. By uncoupling one (sub-)Markov chain into multiple sub-Markov chains, the observer is split (i.e. *uncoupled*) into two observers, each monitoring one part of the registers.

In the case only one set of overlapping processes has to be uncoupled, the uncoupling results in two sub-Markov chains, one accounting for the residuals and one accounting for the set of overlapping processes. The conduct of uncoupling (sub-)Markov chains involves lumping. Contrary to the *reduction* lumping, the *uncoupling* lumping is lossless and therefore reversible via the $\otimes$-composition. The input (sub-)Markov chain is lumped twice. First, all those states are coalesced in which the registers of the residuals are respectively equal and the registers of the overlapping processes differ. Then, all those states are coalesced in which the registers of the overlapping processes are respectively equal and the registers of the residuals differ. The reverse process (i.e. re-coupling) coincides with the composition: $\mathcal{D}_1 = \mathcal{D}_{1,-} \otimes \mathcal{D}_{\pi_5} \otimes \ldots \otimes \mathcal{D}_{\{\pi_9,\pi_{10}\}}$. The example discussed in section 6.5.1 demonstrates the uncoupling of one (product) sub-Markov chain into two (factor) sub-Markov chains in figure 6.15.

Incomplete lumping

In case of bisimilar processes being awarded to different subsystems during the slicing, the bisimilar states they evoke in the full product chain do not occur in the corresponding sub-Markov chains. For instance, consider the BASS example from section 4.3.3. When π_2 and π_3 are awarded to different subsystems, they do not evoke bisimilar states within one sub-Markov chain. Yet, when both sub-Markov chains of π_2 and π_3 are composed, the product chain will carry bisimilar states which then can be lumped. Hence, further bisimilar states possibly arise during the successive re-composition of the locally reduced sub-Markov chains. The *overline* (e.g. $\overline{\mathcal{D}}$) indicates that a Markov chain possibly carries further potential for lumping for that reason. For instance, assume two factor chains to be lumped locally and then recomposed: $\langle \mathcal{D}_i, \mathcal{D}_j \rangle \xrightarrow{\parallel \sim} \langle \mathcal{D}'_i, \mathcal{D}'_j \rangle \xrightarrow{\otimes} \overline{\mathcal{D}'}$. Then the cardinality (i.e. number of states) of $\mathcal{D}'$ is possibly smaller than the cardinality of $\overline{\mathcal{D}'}$ (i.e. $|\overline{\mathcal{D}'}| < |\mathcal{D}'|$). The product chain might contain bisimilar states that do not occur in the factor chains.

---

[6]The relevant information contains the edges of the particular subsystem, the algorithm and the probabilistic influence.

The order in which sub-Markov chains are composed plays an important role as it dictates the size of the maximal intermediate Markov chain. Since state bisimilarity depends on the Hamming distance which depends — in the context of self-stabilizing systems — on the distance between non-processes from the root, it is advisable to privilege composition of sub-Markov chains which subsystems have an equal distance to the root subsystem.

6.3 Decomposition guidelines

Bisimilar processes in the system model evoke bisimilar states in the corresponding transition model. Hence, it is reasonable to put them in the same subsystem. As discussed in paragraph "Reachability vs. Equivalence Class Identification" on page 72, there are indicators that help in identifying bisimilar processes. This section discusses general guidelines exploiting the equivalence class identification to help arriving at a reasonable slicing.

As a consequence of unidirectional fault propagation, cyclic dependencies must be excluded from the system design as discussed in section 6.1. This allows for arbitrary slicing as long as two conditions are fulfilled:

- Each subsystem has at least one overlapping and one residual process, and

- there are no two subsystems with both containing processes that are (directly or indirectly) superior to one or more processes of the respective other subsystem (i.e. exclusion of cyclic dependencies).

Despite these limitations, there are four basic considerations.

Hierarchical cut sets

Lumping is only applicable on bisimilar states in order to preserve the ability to compute the precise LWA. Bisimilar states frequently arise with bisimilar processes that have an equal distance to the root process. Hence, processes with an equal distance to the root process are preferably put into same subsystems.

The reasonable size of a subsystem

To increase the probability that bisimilar processes are put into same subsystems, one objective is to make subsystems as large as possible. The *upper boundary* for the size of subsystems is limited only by the tractability. The reasonable size of subsystems is as large as possible and as small as necessary (i.e. subsystems with only two or less processes are not reasonable).

Non-overlapping sets of overlapping processes

The example in figure 6.9 introduced sets of overlapping processes. When such sets do not mutually overlap, they only have to be computed once. For instance, $\langle \pi_7, \pi_8 \rangle$ is uncoupled from $\mathcal{D}_1$ once and then assigned to the construction of $\mathcal{D}_5$. The same accounts for $\langle \pi_9, \pi_{10} \rangle$ which only has to be uncoupled once to be then included twice, once in $\mathcal{D}_6$ and once in $\mathcal{D}_7$. Finally, it has to be uncoupled from one of them.

Now assume an alternate topology in which π_8 and π_9 are replaced by one process π_8 such that both overlapping sets overlap in that process as shown in figure 6.11.
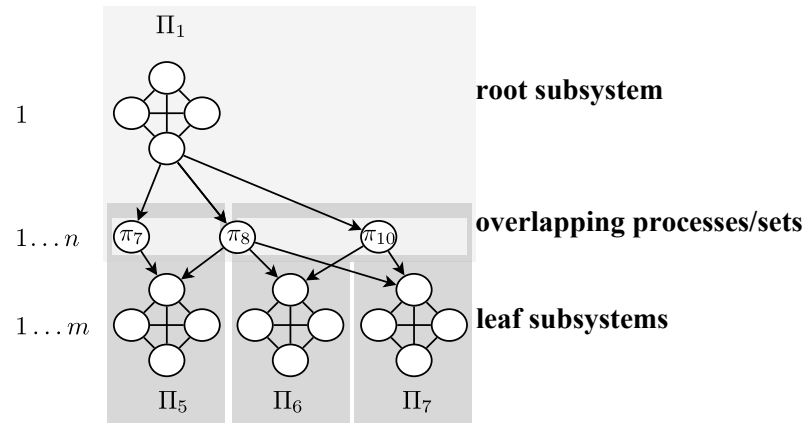
Figure 6.11: Mutually overlapping sets of overlapping processes

Then

- the overlap sets $\langle \pi_7, \pi_8 \rangle$ and $\langle \pi_8, \pi_{10} \rangle$ have to be constructed to be taken into account for the inferior subsystem,

- the overlapping set $\langle \pi_7, \pi_8, \pi_{10} \rangle$ has to be computed to be excluded from the superior subsystem, and

- the *overlapping overlapping*[7] set $\langle \pi_8 \rangle$ needs to be computed to be excluded from all but one of the inferior subsystems.

Overlapping sets that overlap themselves cause further computation steps and should be avoided.

Avoid bisimilar processes in overlaps

It might occur that a (sub-)system is intractable and the only (reasonable) way is to decompose the system such that the minimal overlapping set contains bisimilar processes, for instance when processes π_7 and π_8 in figure 6.9 were bisimilar. Then, both sequences of slicing and lumping are possible. Either the transition model of the superior subsystem is lumped, then uncoupled, and finally the overlapping set is included in the inferior subsystem which is then lumped as shown in figure 6.12(a), or the transition model of the superior subsystem is first uncoupled with the residuals being lumped afterwards while the unlumped overlapping set is included in the inferior subsystem and eventually lumped as shown in figure 6.12(b).

---

[7]The double overlapping is correct here. There are two overlapping sets (vertically between subsystems), i.e. sets of overlapping processes, that overlap (horizontally between overlapping sets). Hence, they are overlapping overlapping sets.

(a) Lump first, uncouple later   (b) Uncouple first, lump later
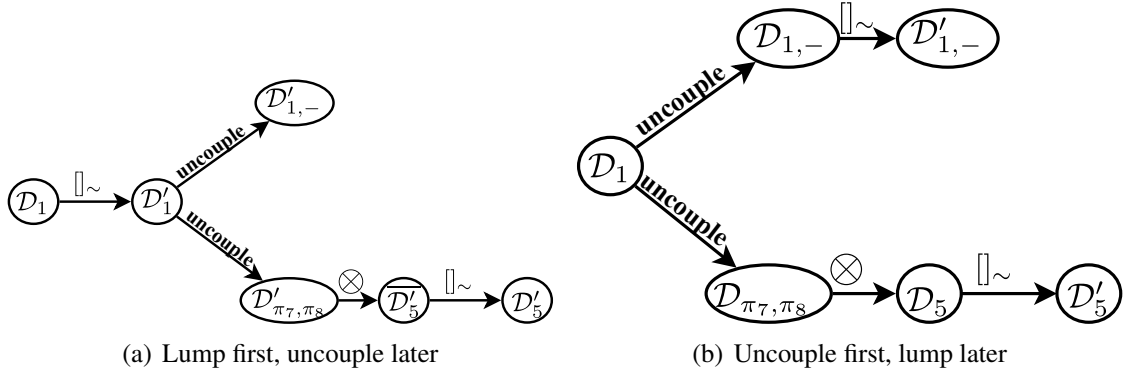
Figure 6.12: Markov chain uncoupling

The benefit of the first procedure is that the transition model of the overlapping set is already minimized. Thus, its subsequent inclusion is less complex. The benefit of the second procedure is that parallelizing the process might benefit from two detached lumping operations that can be executed concurrently. Section 6.6 continues the discussion of decomposition from the perspective of *semi-hierarchic systems*.

6.4 Probabilistic bisimilarity vs. decomposition

Lumping preserves probabilistic bisimilarity with regards to a safety predicate $\mathcal{P}$. This section discusses that the decomposition does not violate the probabilistic bisimilarity either (cf. figure 6.4).

Theorem 6.1 (Decomposition Preserves Bisimilarity)**.**
The decomposition $\tau_k(\mathfrak{S})$ preserves probabilistic bisimilarity between the full DTMC of the original system and the composition with the $\otimes$ operator of the (unlumped) sub-Markov chains of the corresponding subsystems.

Proof 6.1 (Decomposition preserves bisimilarity)**.**
The proof is straightforward. Uncoupling and composing sub-Markov chains with $\otimes$ are reversible and thereby lossless as discussed in paragraph "Uncoupling with $\otimes$" on page 89. Sequentially construction via sub-Markov chains is equal to constructing the full product chain. Both construct the same product chain. Decomposing a system, constructing its sub-Markov chains and composing them with the $\otimes$ operator provides the exact same DTMC as constructing the DTMC directly from the undecomposed system. With both DTMCs being equal, bisimilarity is preserved.

Fault propagation and scheduling

This paragraph points out the importance of fault propagation and scheduling. In hierarchically structured systems, faults propagate in one direction. Systems are decomposed sequentially in the direction of faults propagating from root towards leafs. This marks the first difference to systems with *independent* processes as discussed in paragraph "Related work" on page 78. With independent processes, decomposition is arbitrarily possible.

The second difference arises during the re-composition: Execution semantics must be regarded. With independent processes, an analysis becomes much simpler as they can

commonly be composed in parallel with the Kronecker product. With serial execution semantics, a more sophisticated reasoning about sub-Markov chain composition is necessary. Even when the processes are algorithmically independent, they can still be relying on a mutual central scheduler. The behavior of this scheduler must be accounted for when recomposing sub-Markov chains.

The next step: local lumping

Both lumping and decomposition preserve probabilistic bisimilarity. Thus, the next step is to exploit lumping locally on the sub-Markov chains to avoid the construction of the full product chain. The basic idea is, that from $\mathcal{D} \sim \mathcal{D}'$ follows that also for the subsystems $\forall i : \mathcal{D}_i \sim \mathcal{D}'_i$ holds. Consequently, with $\mathcal{D}'_1 \otimes \ldots \otimes \mathcal{D}'_n = \overline{\mathcal{D}'}$ follows that $\overline{\mathcal{D}'} \sim \mathcal{D}$, based on $\sim$ being a congruence relation with respect to $\otimes$.

6.5 BASS Example

We recall the system model introduced in section 4.3.3 and add the slicing shown in figure 6.14. The source code to reproduce this example are provided in appendix A.5.2. The system model is shown in figure 6.13. It extends the topology shown on page 58 by slicing it in the overlapping process π_4. A probabilistic scheduler $\mathfrak{s}$ selects one of the enabled processes in each computation step. Each process is selected with the same probability. All processes are continuously enabled and serial execution semantics are applied. The processes execute the broadcast algorithm BASS from algorithm 4.6 on page 57.
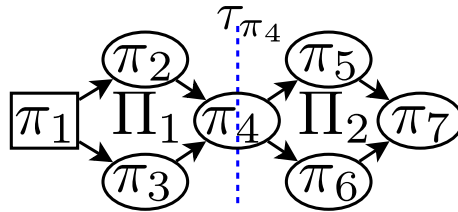
Figure 6.13: The example system - decomposition with $\tau_{\pi_4}(\mathfrak{S})$

The example is small enough to have its full transition model constructed, and also structured ideally to demonstrate decomposition, local lumping and re-composition. Thereby, the example allows to exemplify that probabilistic bisimilarity is preserved throughout the whole process of decomposition, local lumping and re-composition.

While the transition models of systems comprising independent parallel processes can be automatically constructed by sequentially composing the sub-Markov chains, the inter-process dependencies demand that inter-process dependencies are accounted for. Since mutual influence, including algorithmic liveness as well as recovery liveness, makes it inherently challenging to automatically construct a transition model of a hierarchical system, the focus here is not on scalability but on finding the limitations of the approach. The future work section in chapter 8 discusses automatizing the construction of transition models.

The system has two pairs of bisimilar processes: $\pi_2 \sim \pi_3$ and $\pi_5 \sim \pi_6$. As discussed in section 6.3, process π_4 is selected as overlapping process to derive two equally sized and tractable subsystems. The system has a root subsystem $\Pi_1 = \{\pi_1, \ldots, \pi_4\}$, no transient subsystems and one leaf subsystem $\Pi_2 = \{\pi_4, \ldots, \pi_7\}$. All processes are residuals except π_4.

The decomposition pattern

Figure 6.14 shows the decomposition pattern, consisting of five consecutive steps that are iteratively explained in this section.
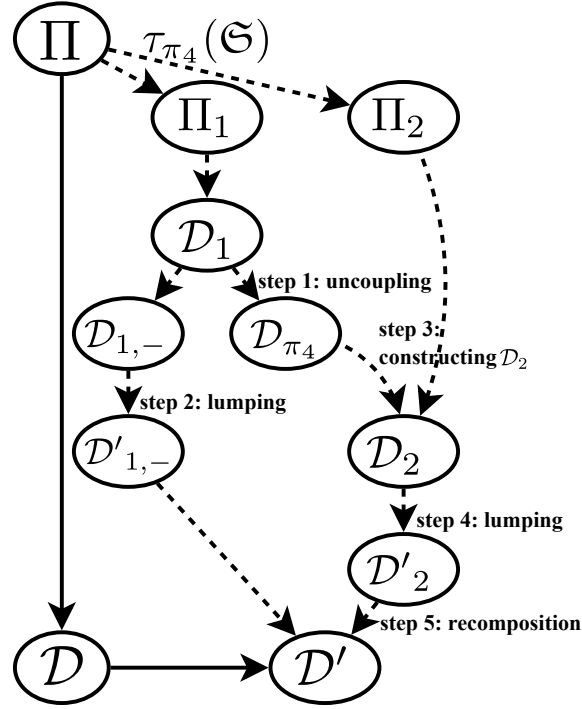

Figure 6.14: Decomposition pattern

The solid arrows show the pattern without decomposition as per figure 6.2 extended by lumping as per figure 6.3. The dotted arrows show the extension by combining decomposition and lumping as per figure 6.5. For now, we disregard the final step that makes the transition model absorbing to compute the LWA. The system is decomposed into $\tau_{\pi_4}(\mathfrak{S}) = \{\{\Pi_1, \Pi_2\}, E, \mathfrak{A}\}$ with $\Pi_1 = \{\pi_1, \ldots, \pi_4\}$ and $\Pi_2 = \{\pi_4, \ldots, \pi_7\}$ and the root sub-Markov chain $\mathcal{D}_1$ constructed according to section 2.4 as if there were no processes π_5, π_6 and π_7. Then, the root sub-Markov chain is uncoupled $\mathcal{D}_1 = \mathcal{D}_{1,-} \otimes \mathcal{D}_{\pi_4}$ into one sub-Markov chain accounting for the residual processes and one sub-Markov chain accounting for the set of overlapping processes. The number of states in the uncoupled root sub-Markov chain $|\mathcal{D}_{1,-}| = 8$ is reduced to $|\mathcal{D}'_{1,-}| = 6$ via lumping. Then, sub-Markov chain $\mathcal{D}_{\pi_4}$ (which is irreducible for obvious reasons) and the remaining processes found the leaf sub-Markov chain $\mathcal{D}_2$ with $|\mathcal{D}_2| = 3^4 = 81$ states. In $\mathcal{D}_2$, 54 states can be lumped to 27 lumps (each representing a pair of states), resulting in $\mathcal{D}'_2$ to comprise $|\mathcal{D}'_2| = 81 - 27 = 54$ states. The recomposed DTMC $\mathcal{D}' = \mathcal{D}'_{1,-} \otimes \mathcal{D}'_2$ (the overline DTMC is skipped as all equivalence classes have already been exploited locally) has a cardinality of $|\mathcal{D}'| = 324$ states which is only half the number of states of $\mathcal{D}$.

6.5.1 Composition method in detail

The composition method comprises five steps shown in figure 6.14. Each following paragraph describes one step. We set the fault probability to $q = 0.05$. Solving Markov models symbolically is also possible. We focus on the numerical evaluation since the symbolic results are very large formulas that are impossible to comprehend and very hard

to compute[8]. The stationary distribution of $\mathcal{D}_1$ to later compute the LWA is shown in table 6.2.

Step 1: Uncoupling the sub-Markov chain $\mathcal{D}_1 \to \mathcal{D}_{1,-} \otimes \mathcal{D}_{\pi_4}$

First, DTMC $\mathcal{D}_1$ is computed shown[9] in figure 6.15(a). The probability that the scheduler selects a process of Π_1 is $\mathfrak{s}_{\Pi_1} = \mathfrak{s}_1 + \ldots + \mathfrak{s}_4 = \frac{4}{7}$. Here, any scheduling probability distribution is feasible and the uniform distribution is selected. Hence, each transition in $\mathcal{D}_1$ is to be multiplied by $\frac{4}{7}$. Then, all *self-targeting* transitions, which are the diagonal entries of the transition matrix $\mathcal{D}_1$, gain the probability mass $1 - \mathfrak{s}_{\Pi_1} = \frac{3}{7}$, which is the probability that a process *outside* Π_1, i.e. in Π_2, is selected for execution. The graphical representation of the transition matrix $\mathcal{M}_1$ is shown in the appendix in figure A.7(a) on page 167.

(a) $\mathcal{D}_1$ (b) $\mathcal{D}_{1,-}$ (top) and $\mathcal{D}_{\pi_4}$ (bottom)
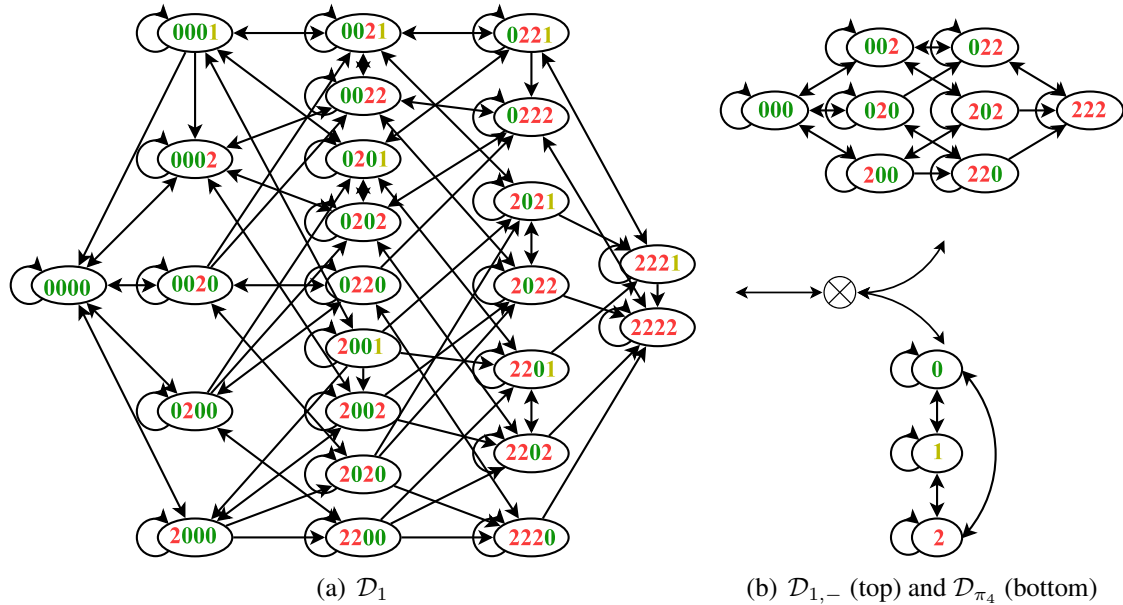
Figure 6.15: Markov chain uncoupling

---

[8]The symbolical computation of this example with MatLab consumes about one week at 2.6 GHz on a Pentium 7, single threaded, and about 48 GBytes of main memory. The numerical computation takes less than a second on the same hardware.

[9]The transition probabilities are omitted from the figure to increase readability.

State	$\langle 0,0,0,0 \rangle$	$\langle 2,0,0,0 \rangle$	$\langle 0,2,0,0 \rangle$	$\langle 0,0,2,0 \rangle$
Probability	0.7238	0.0125	0.0208	0.0208
State	$\langle 0,0,0,2 \rangle$	$\langle 0,0,0,1 \rangle$	$\langle 2,2,0,0 \rangle$	$\langle 2,0,2,0 \rangle$
Probability	0.0469	0.0514	0.0046	0.0046
State	$\langle 2,0,0,2 \rangle$	$\langle 2,0,0,1 \rangle$	$\langle 0,2,2,0 \rangle$	$\langle 0,2,0,2 \rangle$
Probability	0.0008	0.0007	0.0022	0.0063
State	$\langle 0,2,0,1 \rangle$	$\langle 0,0,2,2 \rangle$	$\langle 0,0,2,1 \rangle$	$\langle 2,2,2,0 \rangle$
Probability	0.0308	0.0063	0.0308	0.0048
State	$\langle 2,2,0,2 \rangle$	$\langle 2,2,0,1 \rangle$	$\langle 2,0,2,2 \rangle$	$\langle 2,0,2,1 \rangle$
Probability	0.0005	0.0035	0.0005	0.0035
State	$\langle 0,2,2,2 \rangle$	$\langle 0,2,2,1 \rangle$	$\langle 2,2,2,2 \rangle$	$\langle 2,2,2,1 \rangle$
Probability	0.0077	0.0022	0.0104	0.0037

Table 6.2: Stationary Distribution of $\mathcal{D}_1$

Lumping is applicable to both *uncoupling*, as for instance in paragraph "Step 1: $\mathcal{D}_1 \to \mathcal{D}_{1,-} \otimes \mathcal{D}_{\pi_4}$" as well as *reduction*, as for instance in the next paragraph "Step 2: $\mathcal{D}_{1,-} \xrightarrow{[]_\sim} \mathcal{D}'_{1,-}$". For the uncoupling shown in figure 6.15(b), all states in $\mathcal{D}_1$ that have *the first three digits* in common — for instance where $\langle R_1, R_2, R_3, R_4 \rangle = \langle 0,0,0,0 \rangle$, $\langle 0,0,0,1 \rangle$, or $\langle 0,0,0,2 \rangle$ — are lumped in order to acquire $\mathcal{D}_{1,-}$. Afterwards, all states in $\mathcal{D}_1$ that have *the fourth digit* in common are lumped to acquire $\mathcal{D}_{\pi_4}$. As lumping during uncoupling is lossless, recomposing $\mathcal{D}_{1,-} \otimes \mathcal{D}_{\pi_4}$ is the reverse process resulting in $\mathcal{D}_1$.

The second way to exploit lumping reduces DTMC $\mathcal{D}_{1,-}$ to $\mathcal{D}'_{1,-}$ by lumping probabilistic bisimilar states. The equivalence class $[0,2]_\sim$ containing $\langle R_i, R_j \rangle = \{\langle 0,2 \rangle, \langle 2,0 \rangle\}$ is abbreviated with 2. The double-stroke number indicates the *sum* of values stored in the registers, which is $0 + 2 = 2$ in this case, as introduced in paragraph "The double-stroke alphabet" on page 71. For the analysis, it is regardless which of the processes π_2 and π_3 exactly is corrupted as they behave equally due to same scheduler selection probability, equal fault probability and same position in the system. The corresponding states, for instance $\langle R_1, R_2, R_3 \rangle$ being $\langle 0,0,2 \rangle$ or $\langle 0,2,0 \rangle$, have an equal role in the DTMC.

Sub-Markov chain $\mathcal{D}_1$ is uncoupled into $\mathcal{D}_{1,-}$ shown in table 6.3 — the transition probabilities of the bisimilar states are colored respectively in light and dark gray — and $\mathcal{D}_{\pi_4}$, shown in table 6.4. The stationary distributions of the uncoupled sub-Markov chains are simply the sum of the probability mass in the corresponding states within the original sub-Markov chain. Since the sub-Markov chains are too large to print in numbers, a graphical representation of the transition matrix $\mathcal{M}_{1,-}$ is shown in the appendix in figure A.7(b) on page 167, for $\mathcal{M}'_{1,-}$ in figure A.7(c), and for $\mathcal{M}_{\pi_4}$ in figure A.7(d). In the graphical representations, the default color mapping in MatLab is used where blue means zero and red means one. The transition probabilities in the overlapping sub-Markov chain $\mathcal{D}_{\pi_4}$ are labeled as shown in table 6.4 to later refer to them when $\mathcal{D}_2$ is computed. For the identification of lumpable states, the transition probabilities to all mutual target equivalence classes must be equal.

↓ from/to →	$\langle 0,0,0 \rangle$	$\langle 2,0,0 \rangle$	$\langle 0,2,0 \rangle$	$\langle 0,0,2 \rangle$
$\langle 0,0,0 \rangle$	0.978571	0.007143	0.007143	0.007143
$\langle 2,0,0 \rangle$	0.135714	0.578571		
$\langle 0,2,0 \rangle$	0.135714		0.850000	
$\langle 0,0,2 \rangle$	0.135714			0.850000
$\langle 2,2,0 \rangle$			0.135714	
$\langle 2,0,2 \rangle$				0.135714
$\langle 0,2,2 \rangle$			0.135714	0.135714

↓ from/to →	$\langle 2,2,0 \rangle$	$\langle 2,0,2 \rangle$	$\langle 0,2,2 \rangle$	$\langle 2,2,2 \rangle$
$\langle 2,0,0 \rangle$	0.142857	0.142857		
$\langle 0,2,0 \rangle$	0.007143		0.007143	
$\langle 0,0,2 \rangle$		0.007143	0.007143	
$\langle 2,2,0 \rangle$	0.721429			0.142857
$\langle 2,0,2 \rangle$		0.721429		0.142857
$\langle 0,2,2 \rangle$			0.721429	0.007143
$\langle 2,2,2 \rangle$			0.135714	0.864286

Table 6.3: The transition matrix of $\mathcal{D}_{1,-}$

↓ from/to →	$\langle 0 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$
$\langle 0 \rangle$	$r_4 = 0.982972$	$s_4 = 0.008687$	$t_4 = 0.008341$
$\langle 1 \rangle$	$u_4 = 0.055813$	$v_4 = 0.930721$	$w_4 = 0.013466$
$\langle 2 \rangle$	$x_4 = 0.081422$	$y_4 = 0.023461$	$z_4 = 0.895117$

Table 6.4: The transition matrix of $\mathcal{D}_{\pi_4}$

In table 6.4, the states are labeled to later refer to them in the construction of the inferior sub-Markov chain as discussed above.

Step 2: Lumping the uncoupled root sub-Markov chain $\mathcal{D}_{1,-} \xrightarrow{\langle R_1, [R_2, R_3]_\sim \rangle} \mathcal{D}'_{1,-}$

In $\mathcal{D}_{1,-}$, the two states $\langle 0,0,2 \rangle$ and $\langle 0,2,0 \rangle$ are probabilistic bisimilar and lumped to $\langle 0,2 \rangle$, and analogously are $\langle 2,0,2 \rangle$ and $\langle 2,2,0 \rangle$. The transition matrix of the resulting sub-Markov chain $\mathcal{D}'_{1,-}$ is shown in table 6.5.

↓ from/to →	$\langle 0,0,0 \rangle$	$\langle 2,0,0 \rangle$	$\langle 0,2 \rangle$	$\langle 2,2 \rangle$	$\langle 0,2,2 \rangle$	$\langle 2,2,2 \rangle$
$\langle 0,0,0 \rangle$	0.9786	0.0071	0.0143			
$\langle 2,0,0 \rangle$	0.1357	0.5786		0.2857		
$\langle 0,2 \rangle$	0.1357		0.8500	0.0071	0.0071	
$\langle 2,2 \rangle$			0.1357	0.7214		0.1429
$\langle 0,2,2 \rangle$			0.2714		0.7214	0.0071
$\langle 2,2,2 \rangle$					0.1357	0.8643

Table 6.5: Transition matrix of the root sub-Markov chain $\mathcal{D}'_{1,-}$

Step 3: Constructing the leaf sub-Markov chain $\mathcal{D}_{\pi_4} \otimes |\Pi_2 \setminus \{\pi_4\}| \to \mathcal{D}_2$

The third step takes the uncoupled sub-Markov chain of the overlapping process $\mathcal{D}_{\pi_4}$ and composes it with the processes of Π_2 except π_4 to construct $\mathcal{D}_2$. With $\mathcal{D}_{\pi_4}$ at hand, $\mathcal{D}_2$ can be constructed. In the inferior subsystem Π_2, each process stores either 0, 1 or 2. Therefore — with four processes — the state space of Π_2 comprises $3^4 = 81$ states. The *hybrid* method of using a sub-Markov chain combined with process, scheduling and fault model information is not more complex than constructing a sub-Markov chain, for instance Π_1, from scratch. The transition probability $pr(\overrightarrow{\langle 1, 2, 2, 2\rangle, \langle 2, 2, 2, 2\rangle})$ is exemplarily computed with $w_4 = 0.013466$ provided in table 6.4 as shown in equation 6.8:

$$pr(\overrightarrow{\langle 1, 2, 2, 2\rangle, \langle 2, 2, 2, 2\rangle}) = \frac{1}{4} \cdot w_4 = 0.0033665 \tag{6.8}$$

The transition probability w_4 is given in table 6.4 and multiplied with the probability that π_4 is selected to execute a computation step within Π_2. The probabilities are put into relation to the superior subsystem by multiplying each transition with $\frac{4}{7}$ and adding $\frac{3}{7}$ to the diagonal elements of the sub-Markov chain. Thus, the global probability of the previously computed transition is $0.0033665 \cdot \frac{4}{7} = 0.00192374$. Other transitions where R_4 changes are computed analogously. Transitions between states where R_4 remains unchanged are computed analogously to $\mathcal{D}_1$. Table 6.4 is not required for their computation. The graphical representation of the transition matrix $\mathcal{M}_2$ is shown in the appendix in figure A.7(e) on page 167.

Step 4: Lumping the leaf sub-Markov chain $\mathcal{D}_2 \xrightarrow{\langle R_4, [R_5, R_6]\sim, R_7\rangle} \mathcal{D}_2'$

Reducing $\mathcal{D}_2 \to \mathcal{D}_2'$ offers 27 probabilistic bisimilar state pairs to be lumped. The sets can informally be described as pairs of states, where R_4 and R_7 store equal values, while R_5 and R_6 store unequal values, and each state's R_5 is equal to the mutual other state's R_6. For instance, states $\langle 0, 0, 2, 2\rangle$ and $\langle 0, 2, 0, 2\rangle$ can be lumped to $\langle 0, 2, 2\rangle$. For the computation of LWA, the information which of the registers R_5 and R_6 actually is corrupted is redundant. Knowing that *one* of them is defective suffices to compute LWA. The following pattern defines the sets (i.e. pairs in this case) of probabilistic bisimilar states formally:

The identification of bisimilar states follows the proceeding shown in figure 6.16. States of $\mathcal{D}_2$ are of the form $\langle R_4, R_5, R_6, R_7\rangle$. States

- $\langle x, 0, 1, y\rangle$ and $\langle x, 1, 0, y\rangle$ form $\langle x, \mathbb{1}, y\rangle$,

- $\langle x, 0, 2, y\rangle$ and $\langle x, 2, 0, y\rangle$ form $\langle x, \mathbb{2}, y\rangle$, and

- $\langle x, 1, 2, y\rangle$ and $\langle x, 2, 1, y\rangle$ form $\langle x, \mathbb{3}, y\rangle$.

Notably, the lump notation here is unambiguous as there are no two probabilistic bisimilar states in which both R_5 and R_6 store the value 1. A pair of states $s_i = \langle R_4^i, R_5^i, R_6^i, R_7^i\rangle$ and $s_j = \langle R_4^j, R_5^j, R_6^j, R_7^j\rangle$ is probabilistic bisimilar if

$$s_i = \quad \langle \quad R_4, \quad R_5, \overset{\longleftrightarrow}{\neq} R_6, \quad R_7 \quad \rangle$$

$$s_i \sim s_j : \qquad\qquad = \quad \neq \;\; = \;\; \neq \qquad =$$

$$s_j = \quad \langle \quad R_4, \quad R_5, \underset{\neq}{\longleftrightarrow} R_6, \quad R_7 \quad \rangle$$
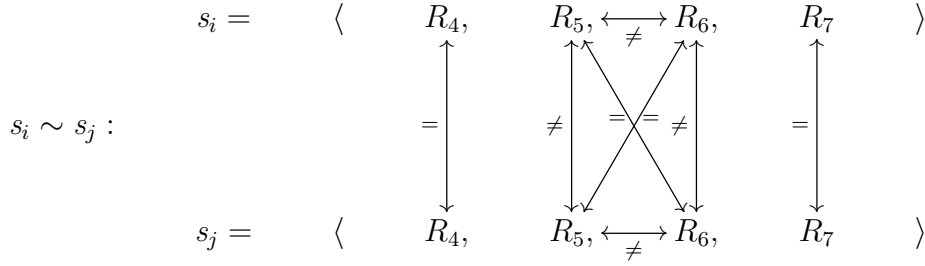
Figure 6.16: Equivalence Class Identification in $\mathcal{D}_2$

The first condition demands both registers R_4 to be equal and analogously for both R_7 registers. The second condition demands the registers R_5 and R_6 to be different within each tuple. The third condition demands register R_5 to be equal to register R_6 of the mutually other state. After the identification of the probabilistic bisimilar pairs, the affected transitions are lumped. The lumping of the transition $pr(\overrightarrow{\langle 1, \mathbb{1}, 0 \rangle, \langle 0, \mathbb{1}, 0 \rangle})$ is presented exemplarily in figure 6.17.
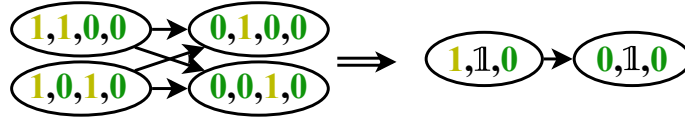
Figure 6.17: Reduction example

The first lump $\langle 1, \mathbb{1}, 0 \rangle$ comprises states $\langle 1, 1, 0, 0 \rangle$ and $\langle 1, 0, 1, 0 \rangle$. The second lump $\langle 0, \mathbb{1}, 0 \rangle$ comprises states $\langle 0, 1, 0, 0 \rangle$ and $\langle 0, 0, 1, 0 \rangle$. While some transition probabilities are zero:

- $pr(\overrightarrow{\langle 1, 1, 0, 0 \rangle, \langle 0, 0, 1, 0 \rangle}) = 0$

- $pr(\overrightarrow{\langle 1, 0, 1, 0 \rangle, \langle 0, 1, 0, 0 \rangle}) = 0$

due to serial execution semantics as discussed in paragraph "Hamming Distance" on page 15, others contribute to the aggregated transition probability:

- $pr(\overrightarrow{\langle 1, 1, 0, 0 \rangle, \langle 0, 1, 0, 0 \rangle}) = u_4 \cdot \mathfrak{s}_4 \cdot \frac{4}{7}$

- $pr(\overrightarrow{\langle 1, 0, 1, 0 \rangle, \langle 0, 0, 1, 0 \rangle}) = u_4 \cdot \mathfrak{s}_4 \cdot \frac{4}{7}$.

The variable u_4 is the transition probability of R_4 changing its value from 1 to 0 as shown in table 6.4, and $\mathfrak{s}_4 = \frac{1}{4}$ is the execution probability of π_4 within the subsystem Π_2. As discussed in paragraph "Step 1", the distinction between the possibilities that either a process *in* the sub-Markov chain $\mathcal{D}_2$ is selected for execution, or a process *outside* is selected — that is within $\mathcal{D}_{1,-}$ — is important. This distinction is regarded *before* lumping. Hence, the transition probabilities are multiplied with $\frac{4}{7}$.

$\downarrow$ from/to $\rightarrow$	$\langle 0, 1, 0, 0 \rangle$	$\langle 0, 0, 1, 0 \rangle$
$\langle 1, 1, 0, 0 \rangle$	$u_4 \cdot \pi_4 \cdot \frac{4}{7}$	0
$\langle 1, 0, 1, 0 \rangle$	0	$u_4 \cdot \pi_4 \cdot \frac{4}{7}$

Table 6.6: Example Transition Lumping of Transition $pr(\overrightarrow{\langle 1, \mathbb{1}, 0 \rangle, \langle 0, \mathbb{1}, 0 \rangle})$

With steady state probabilities $pr_\Omega(\langle 1,1,0,0\rangle) = pr_\Omega(\langle 1,0,1,0\rangle) = 0.002557259339314$ and the above transition probabilities, equation 5.6 from page 67 computes the lumped transition probability shown in equation 6.9 according to figure 6.17:

$$pr(\overrightarrow{\langle 1,\mathbb{1},0\rangle, \langle 0,\mathbb{1},0\rangle}) = \frac{pr(\overrightarrow{\langle 1,1,0,0\rangle, \langle 0,1,0,0\rangle}) \cdot pr_\Omega(\langle 1,1,0,0\rangle) + pr(\overrightarrow{\langle 1,0,1,0\rangle, \langle 0,0,1,0\rangle}) \cdot pr_\Omega(\langle 1,0,1,0\rangle)}{pr_\Omega(\langle 1,1,0,0\rangle) + pr_\Omega(\langle 1,0,1,0\rangle)}$$
(6.9)

The other transitions are computed analogously and $\mathcal{D}_2'$ is constructed. The graphical representation of the transition matrix $\mathcal{M}_2'$ is shown in the appendix in figure A.7(f) on page 167.

Step 5: Re-composition $\mathcal{D}' = \mathcal{D}_{1,-}' \otimes \mathcal{D}_2'$

With $\mathcal{D}_{1,-}'$ and $\mathcal{D}_2'$ at hand, $\mathcal{D}'$ is composed. Notably, both reduced sub-Markov chains $\mathcal{D}_{1,-}'$ and $\mathcal{D}_2'$ execute computation steps parallel as their probabilities have been weighted. For re-composition, each transition in $\mathcal{D}_{1,-}'$ is multiplied with each transition in $\mathcal{D}_2'$. The coordinates are labeled row i and column j in $\mathcal{D}_{1,-}'$, and k and l in $\mathcal{D}_2'$ respectively. Notably, transitions between states that differ in more than one register must be dealt with separately to cope with serial execution semantics. Algorithm 16 computes the re-composition for serial execution semantics. The graphical representation of the transition matrix $\mathcal{M}'$ is shown in the appendix in figure A.7(g) on page 167.

The final step to transform $\mathcal{D}'$ to $\mathcal{D}_{\mathrm{LWA}}'$ to compute LWA — the stationary distribution is known — is to set[10] the transition probability $\mathcal{M}'(1,1) := 1$, and $\forall m, 1 < m \leq 324 : \mathcal{D}'(1,m) := 0$ as discussed in section 4.3.2 (cf. also [Müllner and Theel, 2011, ch.4]). Then, the legal state is absorbing. The computed LWA coincides with the LWA computed with the full product chain as depicted in figure 4.9 on page 61.

6.5.2 Example interpretation

We start with discussing the benefits of the decomposition and then reason about the results of the example.

The decomposition

The goal was to simplify the computation of the LWA. Instead of 648 states in $\mathcal{S}$, the decomposition method was able to construct a probabilistic bisimilar DTMC where $\mathcal{S}'$ contains only half as many states. The example demonstrated the challenges of decomposing hierarchic systems and presented an approach for decomposition with *overlapping processes*. In the second half of the procedure, the re-composition, execution semantics have been found to play an important part. Contrary to direct dependency among processes via fault propagation, execution semantics let processes depend indirectly. The $\otimes$ operator has been introduced as replacement for the Kronecker product in cases where maximal parallel execution semantics are not applied. Both hierarchic ordering and serial execution semantics have been discussed in this present example to explain, how LWA can be computed for hierarchically structured systems with less than maximal parallel execution semantics.

---

[10]$\mathcal{M}'(1,1)$ is the transition in the first row and the first column in $\mathcal{D}'$, i.e. $pr(\overrightarrow{\langle 0,0,0,0,0,0\rangle, \langle 0,0,0,0,0,0\rangle})$.

The LWA

The LWA vector serves three purposes:

1. In case the system under consideration is supposed to obtain a certain least amount of availability, the LWA vector can be exploited to acquire the associated amount of time that is required to meet the demand.

2. When the system is allowed a maximal distinct number of computation steps, the LWA vector can be exploited to determine the achievable availability for that amount of temporal redundancy.

3. In case multiple solutions to the same problem are applicable, LWA can be a valuable quantification of fault tolerance to select the *optimal* solution as discussed in section 4.5.

Despite these, another interesting question arises: What are the most *critical* states, from which the system is most unlikely to recover *in time*?

The *probability mass drain* exposes two equivalence classes to discuss this question. Figure 6.18 shows the probability mass over illegal equivalence classes and time, similar to figure 4.10 on page 61 did for states and time.
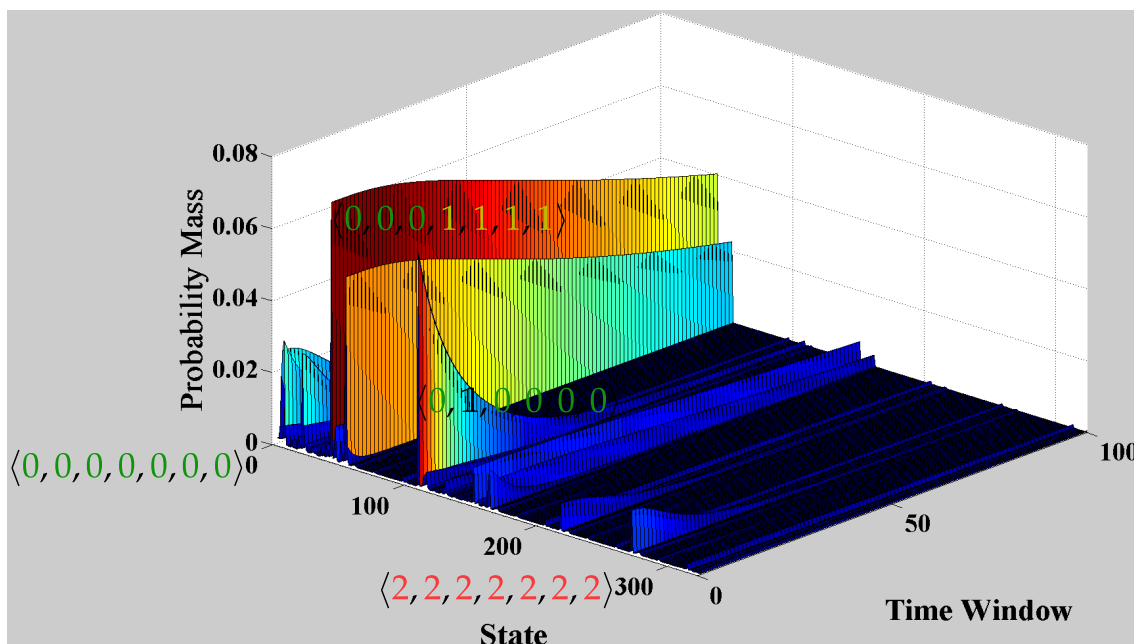
Figure 6.18: Probability mass drain

To increase readability, the y-axis showing the probability mass in each state for each time window is cropped at 0.08, which is a little more than the maximal probability mass an unsafe state contains in the limit. Equivalence classes $\langle 0, 0, 0, 1, 1, 1, 1 \rangle$ and $\langle 0, \mathbb{1}, 0, 0, 0, 0 \rangle$ both contain not only the most, but also a similar amount of probability mass in the limit.

Although initially — from the limit onwards — equipped with a similar amount of probability mass[11], state $\langle 0, \mathbb{1}, 0, 0, 0, 0 \rangle$ looses its probability mass rapidly compared to state $\langle 0, 0, 0, 1, 1, 1, 1 \rangle$ as shown in figures 6.19(a) and 6.19(b).

(a) State $\langle 0, 0, 0, 1, 1, 1, 1 \rangle$ (b) State $\langle 0, \mathbb{1}, 0, 0, 0, 0 \rangle$
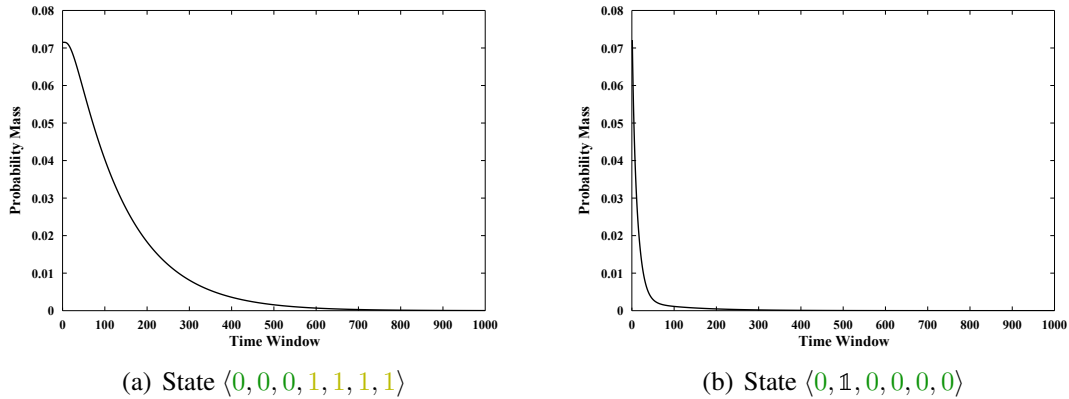
Figure 6.19: Comparing states

The motivation to compute LWA in the first place was to find the amount of time required to achieve a desired probability for a non-masking fault-tolerant system to mask faults as discussed in chapter 3. Knowing about the probability mass drain of *all* states (and lumps) allows yet for much more. Some systems offer the possibility for certain states to be either prevented or instantly repaired. One applicable method is *snap stabilization* [Tixeuil, 2009, Delporte-Gallet et al., 2007]. It provides a functionality for instantaneous recovery by (re-)setting distinct states directly to a legal value. When searching for states to apply such targeted counter measures, state $\langle 0, 0, 0, 1, 1, 1, 1 \rangle$ is obviously a far more suitable target than state $\langle 0, \mathbb{1}, 0, 0, 0, 0 \rangle$. Either *preventing* state $\langle 0, 0, 0, 1, 1, 1, 1 \rangle$ to gather probability mass or employing targeted counter measures to *drain* this state at a faster pace both seem desirable targets. The set of equivalence classes can be sorted according to their probability mass for each time step in this manner. Such a list tells, which equivalence classes withhold the most probability mass at a certain time. When the upper boundary for w and the number of distinct stated to evade or drain are known, it is simple to look at that list at that time point and to pick the top number of states that can be evaded to produce an optimal result regarding a system's fault tolerance.

6.6 Decomposability - A matter of hierarchy

In paragraph "Related work" on page 78 it was pointed out, that a parallel composition with the Kronecker product is not applicable for hierarchic systems. Afterwards, this chapter focused on the decomposition of hierarchic systems. Yet, as discussed in section 6.1, there are systems in which every process depends on every other process, called *heterarchic* systems. When all process rely on each other, decomposition becomes a lot less promising. With independent and hierarchical systems being ideal for decomposition and heterarchic systems being the *worst case* for decomposition, this section is dedicated to discuss the possibilities for decomposition of systems that are between the extremes. It

---

[11]The probabilities in the limit are $pr_\Omega(\langle 0, 0, 0, 1, 1, 1, 1 \rangle) = 0.0715034677782571$ and $pr_\Omega(\langle 0, \mathbb{1}, 0, 0, 0, 0 \rangle) = 0.0721206997887467$.

determines a possible classification of semi-hierarchical systems and proposes individual approaches on how to accomplish decomposition in each identified case.

Semi-hierarchic systems are — like redundancy — hierarchic either in space or in time or in both. In terms of the Bernstein conditions discussed on page 86, semi-hierarchic systems violate the third condition only locally or temporarily while otherwise violating only the first condition or none at all.

6.6.1   Classes of semi-hierarchical systems

Three system classes between purely hierarchical and purely heterarchical, referred to as *semi-hierarchical*, can be distinguished. In semi-hierarchical systems, *dependability slices* can be identified that are *locally* of *temporarily* hierarchic. If not the whole system is decomposable, possibly decomposition within dependability slices is applicable.

Contrarily, it is not possible for systems to be globally heterarchical and hierarchical otherwise. Assume a hierarchic subsystem that is mutually reliant with another subsystem. Then, every process within each subsystem relies on the other subsystem and as a consequence indirectly on itself. Thereby, every process relies on every other process and there cannot be local hierarchies.

The first type of semi-hierarchy is *locally* heterarchical and hierarchical otherwise, the second is *temporarily* heterarchical, and the third class accounts for dynamic topologies.
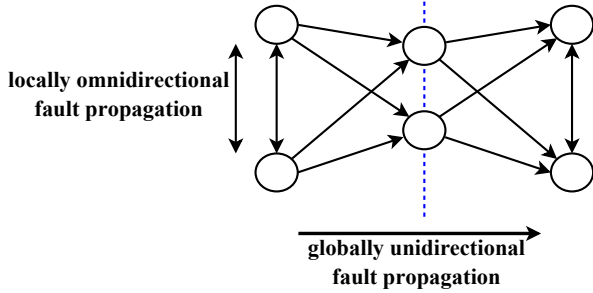
Type	Description
local	One class of semi-hierarchic systems contains *locally* heterarchic subsystems like dependability cycles, which *globally* influence each other *hierarchically*. Such systems can be sliced into locally heterarchic subsystems, but the subsystems cannot be sliced any further. The following figure provides an example.

temporal	A temporally semi-hierarchic system can switch between phases of hierarchy and heterarchy. Then, each interval of hierarchy is called in *epoch*[13].
	For instance, consider a self-stabilizing leader election algorithm (LE) [Fischer and Jiang, 2006, Nesterenko and Tixeuil, 2011] to execute when there is no elected leader to act as root process, and that another self-stabilizing work-algorithm requiring a designated root process executes otherwise. Each case of elected leader is then one system instance for the work algorithm. During phases of leader election — that is between epochs — the processes are mutually dependent.
dynamic	Despite static topologies, systems with dynamic topologies face a similar problem like temporally semi-hierarchic systems. Processes and edges connecting them do not necessarily need to be static. When processes enter and leave a dynamic topology, each possible topology instantiation is to be regarded with its own transition model. The topologies can then be linked with the probabilities for processes to enter or leave the system like in the case of temporal semi-hierarchy.

Table 6.7: Classifying categories of local heterarchies in globally hierarchic systems

While the first case is straight forward, the second and third case introduce a new issue. They distinguish cases — elected leader and topology instantiation — and each case requires to be regarded with its own transition model. Furthermore, they possibly provide transition probabilities connecting the *case transition models*, meaning one transition model for each probability distribution. Consider for instance the TLA from sections 2.5 and 4.3.2. For simplicity, assume that only exclusively one of the probabilistic influences of either scheduler or fault model probabilistically switches between three

different[14] probability distributions. Figure 6.20 exemplarily shows that the three case transition models are *uniform* for three different fault probability distributions q_1, q_2 and q_3 in the sense that they span the equal state space and contain positive transition probabilities for the same state tuples. Yet, the particular transition probabilities differ according to the fault probability distributions.
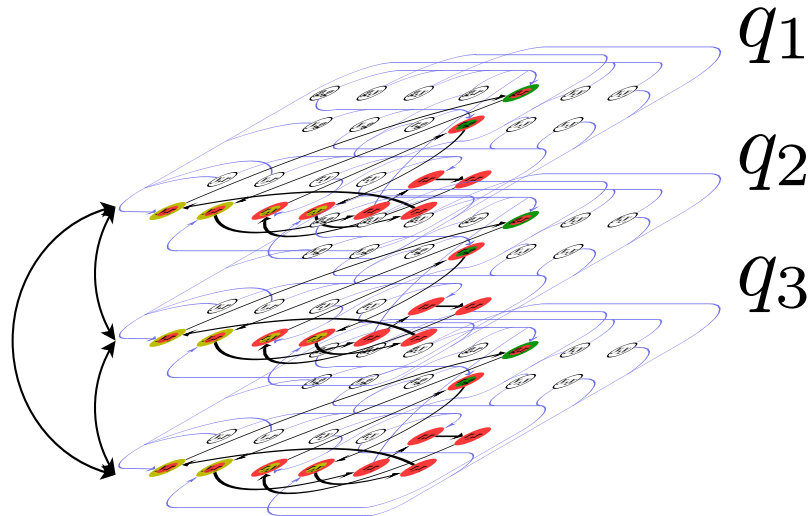

Figure 6.20: Multiple layers of transitional models

The layers are connected via orthogonal transitions[15]. Assume that the availability of the processes can be controlled either discretely or continuously. In the discrete case, layers as depicted in figure 6.20 specify the operation modes of the system that can dynamically adapt its fault tolerance. In the continuous case, the transition model becomes a infinite space stochastic process.

The basic concept of *layers* within the transition model arising with semi-hierarchy possibly provides further leverage for lumping. The desired *uniformity* among the layers of temporally semi-hierarchic systems can be generally achieved via symmetric structures as exemplified in the following paragraph.

6.6.2 Temporal semi-hierarchy and topological symmetry

The case of combining LE with a working algorithm provides another interesting discussion regarding the topology. The system topology is not only important to its decomposability, but also for the lumpability in the analysis of temporally semi-hierarchic systems. For instance, assume a platonic solid — a regular convex polyhedra — like eight processes that are connected like a cube as shown in figure 6.21. During epochs, the processes execute the BASS under uniformly distributed fault and scheduling probabilities. In case a fault initiates the election of a new root, a self-stabilizing LE executes until a leader is elected and a new epoch begins.

---

[14] This possibilities for alternating the example are limited since i) changing the topology would result in transition models with different state spaces and ii) changing the root is not possible in the heterarchic TLA.

[15]Figure 6.20 abstracts the single orthogonal transitions as indicated by the black arrows.
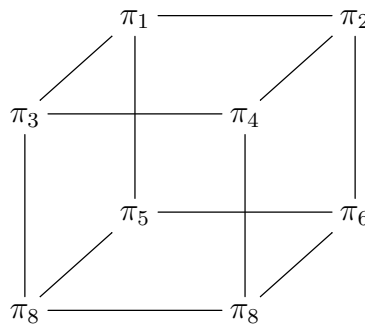
Figure 6.21: Platonic leader election

Irrespective of which process becomes the elected leader, the transition model will in any case be the same. The transition model only requires one additional state (or transition model) to account for the intervals between the epochs. This example demonstrates that highly symmetric systems are promising for being highly lumpable.

6.6.3 Mixed mode heterarchy

This section briefly reasons about how mixed mode systems — these are systems executing multiple algorithms in parallel — forfeit hierarchy and become heterarchic. Assume a distributed system with parallel maximal execution semantics and two algorithms being executed in parallel: For instance, a wireless sensor network in which the sensors propagate measured data to a central process is updated with control strategies by the same central process frequently. The control strategies can include data type (temperature or humidity) or update frequency. Each system is hierarchical for itself. The sensor motes propagate straight towards the center process and the center process sends the updates towards the leafs. The *whole* system utilizes one communication infrastructure for bidirectional communication. Although the systems are *algorithmically independent*, they rely on each other by sharing a common resource. Message congestion caused by one algorithm also blocks convenient routes for the other algorithm. Assuming that message congestion and real time constraints are critical issues, both systems are considered to be dependent. Notably, the communication directions need not necessarily be opposing.

6.7 Summarizing decomposition

This chapter started by classifying independent, hierarchic, semi-hierarchic and heterarchic systems. This work's context motivated to focus on hierarchic systems. After introducing the necessary formalisms and outlining basic guidelines for decomposition, the BASS example was continued to discuss how decomposition and lumping can be practically combined. Interpreting the results provided for an insightful discussion. Finally, a pathway to discussing semi-hierarchic systems was proposed by classifying them and proposing approaches for each class individually. The following chapter uses these findings to focus on two properties exemplarily: i) the complexity of parallel composition to contrast the intricacies of hierarchic decomposition, and ii) a small semi-hierarchic system with execution semantics that are neither serial nor maximal parallel to show the benefits of hierarchy.

7. Case studies

This section provides two case studies to demonstrate the practical utility of the presented methods and concepts, and to point out technical difficulties that could not be addressed before. The first case study focuses on parallel systems comprising independent processes, similar to the work discussed in paragraph "Related work" on page 78, to compare parallel independent and hierarchical systems regarding their composability. The second case study considers relaxing several assumptions like concerning execution semantics and hierarchical dependencies. Both case studies start with discussing how a DTMC can be derived from the real world model in general before demonstrating a concrete example.

7.1   Thermostatically controlled loads in a power grid

The methods that have been developed in the present thesis are applied within the project *Modeling, Verification and Control of Complex Systems* (MoVeS), funded by the European Commission under grant FP7-ICT-2009-257005. The goal of this case study is to determine the risk of voltage peaks in a power grid. Such voltage peaks occur when the accumulated load demanded by the consumers changes too fast, that is, when too many consumers simultaneously either increase of simultaneously decrease their energy demand. In this example we consider the load to be caused by cooling systems that are controlled by thermostats. The example is based on a case study about *thermostatically controlled loads* (TCL) presented by Callaway [Callaway, 2009] in 2009. The scenario itself is based on the temperature model proposed by Malhame and Chong [Malhamè and Chong, 1985] in 1985 and was later extended by Koch et al. [Koch et al., 2011] in 2011. Parts of this section are published in [Kamgarpour et al., 2013].

The TCL model

Consider a set of homogeneous houses in a warm region. While the ambient temperature θ_a outside is constantly $32\,°\mathrm{C}$, the desired set indoor temperature θ_s is $20\,°\mathrm{C}$. A thermostat

controls the cooling system in the house. It turns on when the temperature reaches the upper bound of the hysteresis $\delta = 0.5\,°C$, which is $20.5\,°C$, and it turns off when reaching the lower boundary which is $19.5\,°C$.

The deadband

Similar to defining safety to demarcate legal from illegal states, temperature bands can be used to specify *comfort zones* in which the thermostat should operate. Consider that the thermostat has a latency of one time step and measures the temperature at discrete evenly distributed time points utilizing a *bang-bang* control [Sonneborn and van Vleck, 1964]. A *bang-bang* control is a simple on/off switch turning the cooling system on when it is too hot, and off when it is too cold. It is reasonable to specify the comfort zones according to how far the actual temperature deviates from θ_s. The system is

- in a *legal state* within $[19.5, 20.5]$,

- a switching should occur within a *reasonable interval*, that is, not too long after the deadband is left,

- an undesired state is reached beyond that interval, when switching did not occur *timely*.
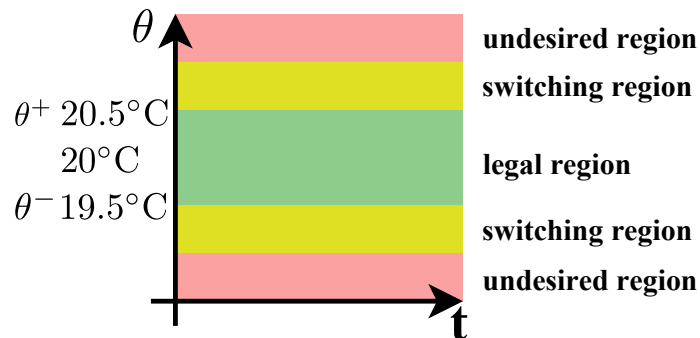
Figure 7.1: Specifying legal and undesired states

This shows how classification of fault, error and failure discussed on page 9 can be mapped onto temperature intervals.

Temperature progress

The following equation describes the temperature progress:

$$\theta(t+1) = \underbrace{a\theta(t)}_{i)} + \underbrace{(1-a)(\theta_a - m(t)R \cdot P)}_{ii)} + \underbrace{g(t)}_{iii)} \text{ [Callaway, 2009, } p.8] \qquad (7.1)$$

The equation[1] reads as follows: the temperature in the next time step is i) the temperature of the current time step plus ii) the temperature progress depending on whether the

---

[1]The equation has been adapted from [Callaway, 2009, p.8] in the context of this thesis. For instance, the original version uses w instead of g as noise term. To avoid ambiguity with the window size, equation 7.1 shows $g_i(t_n)$ as noise term.

thermostat is turned on or off plus iii) some noise. Parameter a "governs the thermal characteristics of the thermal mass and is defined as $a = \exp(-h/CR)$" [Callaway, 2009] with h being the duration of a time step measured in seconds, C being the thermal capacitance measured in $kWh/\,^\circ C$ and R being the thermal resistance measured in $^\circ C/kW$. The switch m is defined as follows:

$$m_i(t_{n+1}) = \begin{cases} 0, & \theta(t) < \theta_s - \delta = \theta_- \\ 1, & \theta(t) > \theta_s + \delta = \theta_+ \\ m(t) & \text{otherwise [Callaway, 2009, } p.9] \end{cases} \tag{7.2}$$

Parameter P described the energy transfer rate to or from the thermal mass measured in kW. The term $g(t)$ is a noise term. Table 7.1 shows the standard parameters used by Callaway [Callaway, 2009]:

Parameter	Meaning	Standard value	Unit
R	average thermal resistance	2	$^\circ C/kW$
C	average thermal capacitance	10	$kWh/\,^\circ C$
P	average energy transfer rate	14	kw
η	load efficiency	2.5	
θ_s	temperature set point	20	$^\circ C$
δ	thermostat hysteresis	0.5	$^\circ C$
θ_a	ambient temperature	32	$^\circ C$

Table 7.1: Model parameters

Parameter η is required to describe the total power demand: $y(t+1) = \sum_{i=1}^{N} \frac{1}{\eta} P \cdot m(t+1)$. The parameter describes the efficiency "and can be interpreted as the coefficient of performance" [Callaway, 2009].

Deterministic execution

To determine the influence of each single parameter, the system execution is evaluated at first without noise, that is, without part iii) in equation 7.1. Then, the system is deterministic without probabilistic influence. The corresponding implementation in iSat [Fränzle et al., 2007] is provided in the appendix A.5.7 on page 172.
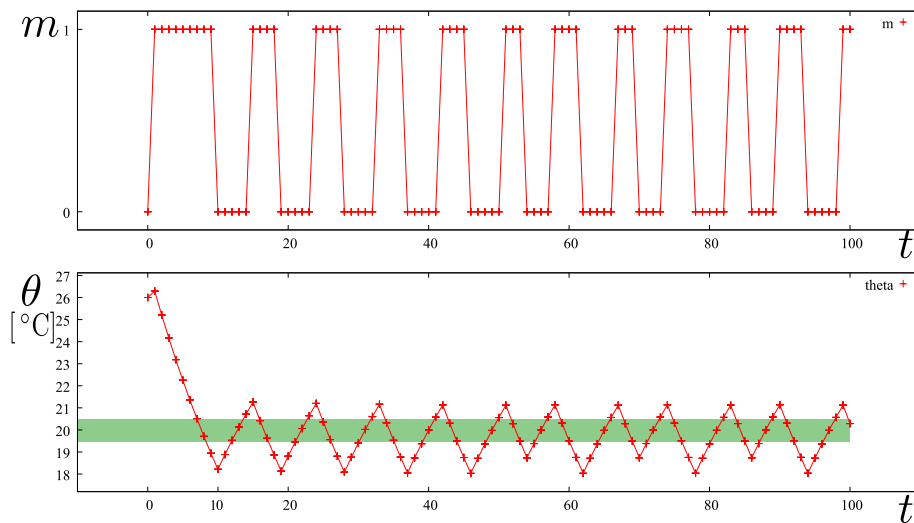
Figure 7.2: The TCL model executing with standard parameters

We first describe the initial behavior and then the behavior in the limit. The upper graph in figure 7.2 shows the status of the switch and the lower graph shows the temperature evolving over time. Initially, the system detects that the temperature is too high and initiates cooling one time step later. At time step 8, it enters the deadband for the first time — at time step 7 it is just above the hysteresis — and continues cooling until time step 9.

- The system requires ten time steps to reach the lower boundary of the hysteresis for the first time,

- the switch is turned off for (alternatingly) three or four steps,

- the switch is turned on for (alternatingly) two or three steps, and

- the repetitive switching cycle shown in Figure 7.3 occurs the first time at time instant 44 and persists at least until time step 100. It even holds until time step 1000, not depicted in the graph, so that the assumption that the cyclic behavior is stable is justified.
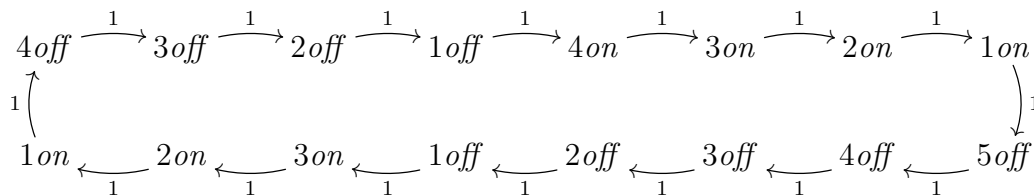
Figure 7.3: Repetitive cycle

Each vertex is labeled *number status*, referring to the number of steps the system will remain in the status. The deterministic setting without noise allows to understand how the single parameters influence the equation. Therefore, we repeat the same setting but change each parameter one at a time, amplifying it by a factor of ten compared to the standard parameters from table 7.1, except for the parameters altered in figures 7.4(e)

and 7.4(f), which are amplified by adding $10\,°C$ in figure 7.4(e) and subtracting $10\,°C$ in figure 7.4(f).

(a) $R = 20\,°C/kW$

(b) $P = 140kw$

(c) $C = 100kWh/\,°C$

(d) $\delta = 5\,°C$

(e) $\theta_a = 42\,°C$

(f) $\theta_s = 10\,°C$

(g) $\eta = 25$
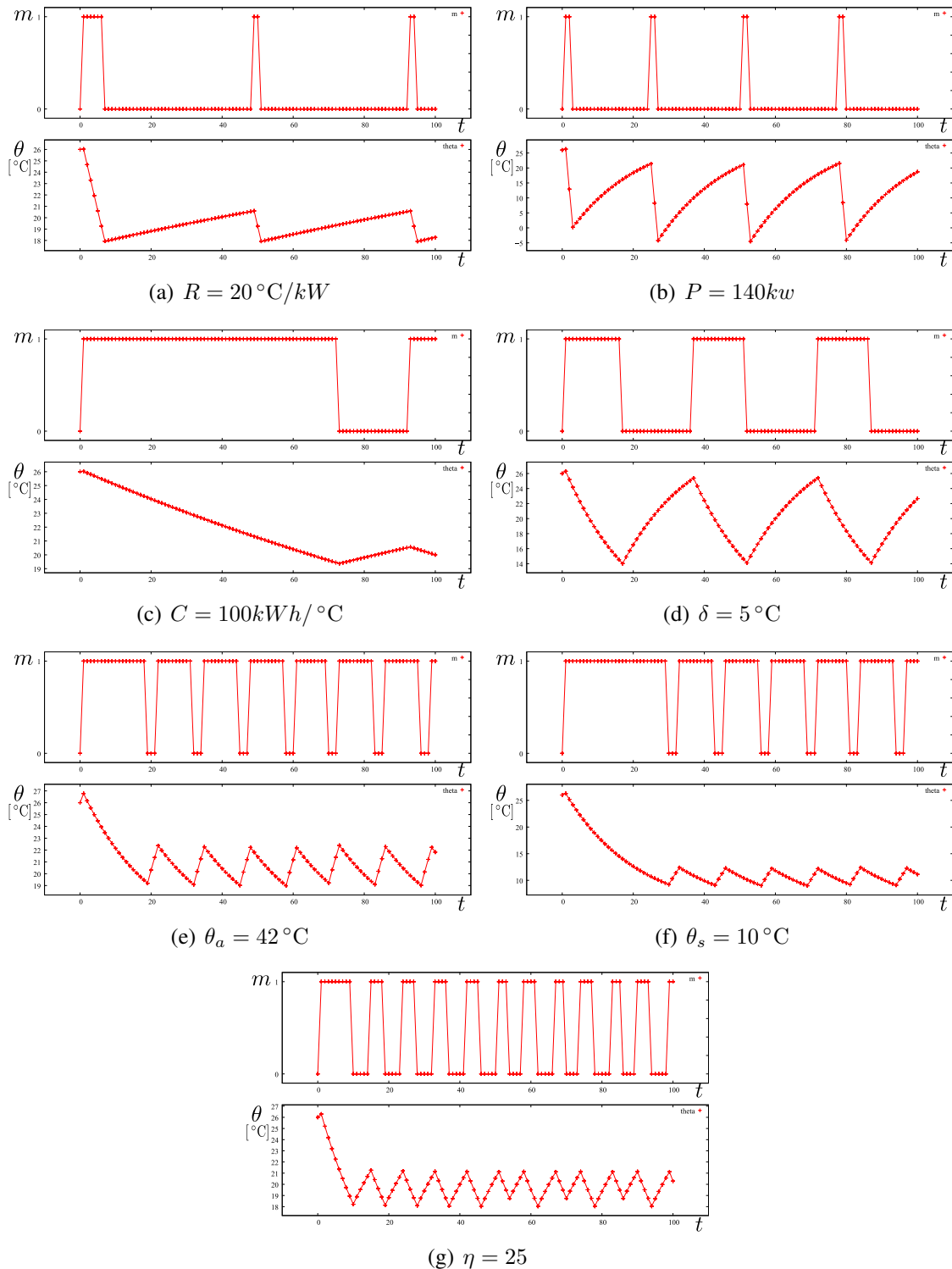
Figure 7.4: Deviating parameters

When the isolation of the house via parameter R is increased, it heats up at a far slower pace as shown in figure 7.4(a). Furthermore, the cooling process is more efficient. The switching delay forces the system to even cool below the *safety* threshold as the temperature reaches below $18\,°C$. Figure 7.4(b) shows that amplifying the cooling power via

P cools the house down rapidly. With the delay of one time step given, the cooling device freezes the house even below $0\,°\mathrm{C}$. In case the thermal capacitance is increased — imagine for instance the house filled with a liquid instead of air — via parameter C, both cooling and heating phases are slowed down as shown in figure 7.4(c). If the deadband is relaxed via parameter δ as shown in figure 7.4(d), the cooling and heating phases take longer as well. Since it would be unreasonable to amplify the ambient temperature via θ_a beyond a certain point, $10\,°\mathrm{C}$ are added instead of multiplying by a factor of 10. As shown in figure 7.4(e), the heating phases are shortened and the cooling phases are extended. The setting in depicted in figure 7.4(f) lowers the set point θ_s to $10\,°\mathrm{C}$ which also shortens the heating phase and flattens the graph. Amplifying the load efficiency via η by a factor of ten as shown in figure 7.4(g) has almost no effect.

Adding noise

When the TCL model without noise is explored, the system executes along one deterministic execution trace. By adding a general noise term, like the last part in equation 7.1, the transition model becomes Markovian. The execution traces then *spread over time* as exemplarily shown in figure 7.5 by Koch et al. [Koch et al., 2011].
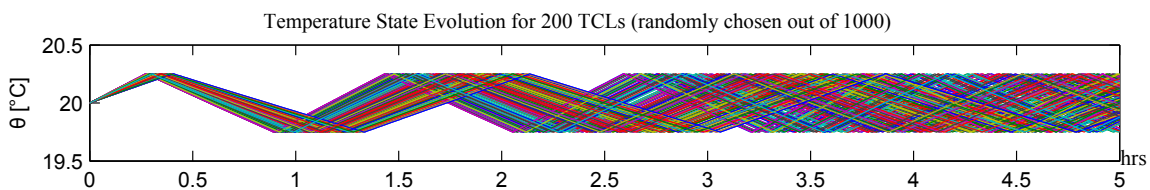
Figure 7.5: Temperature state evolution via simulation by Koch et al. [Koch et al., 2011, p.3]

In this setup, 200 households execute in parallel with noise added, thus reaching the deadband boundaries at different times. While they are all initially in the same state, their progress differs such that after about three hours it seems as if all synchronicity is lost. One time step lasts ten seconds in this example.

Binning

The method to compute a window property like LWA is based on system and probabilistic influence to be translated into a DTMC. An intermediate step to finally acquire a DTMC from the TCL scenario is the discretization of the continuous temperature domain. A discretization in this context is commonly known as *binning* [Callaway, 2009, Koch et al., 2011]. The temperature domain is partitioned into — in this case equally sized — *bins*.

The probabilistic execution traces reach a bin with a certain probability in the next time step. The progress of each household along the temperature domains — one domain for m being off and one for m being on — can be formally be described with a DTMC as pictured in figure 7.6.
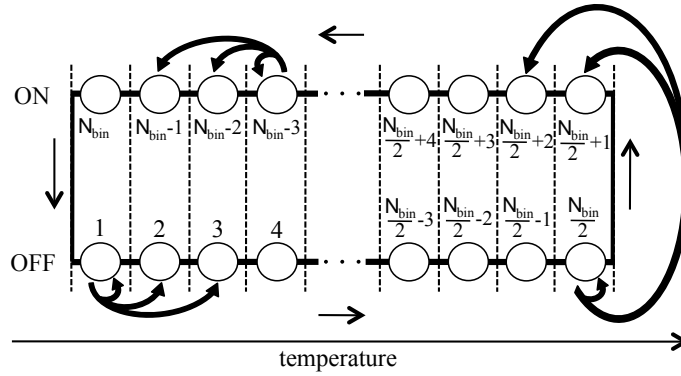
Figure 7.6: The state bin transition model by Koch et al. [Koch et al., 2011, p.2]

The figure shows how the temperature domain is binned for both on and off states of m, and that the transition probabilities can be computed for each state tuple. The TCL example points out the limitations of deriving precise transition probabilities analytically. Transition probabilities are often derived via approximate methods like simulation or sampling. In this case, binning is an abstraction introducing an error. The coarser the bin are, the greater becomes the abstraction error. Soudjani and Abate [Soudjani and Abate, 2013a, Soudjani and Abate, 2013b, Soudjani and Abate, 2013c] currently work on methods to compute the error that is introduced by the abstraction. Notably, they propose a method to *directly* compute the transition probabilities in a product chain of multiple housings, contrary to the sequential construction of the lumped product Markov chain that is discussed in this section.

The analytic methods proposed in the previous chapters rely on the quality of the provided probabilities. The discussion of power grids addressed, that determining this quality is important. Furthermore, it showed, that safety can be formulated and a DTMC can be constructed to evaluate the safety over time. The remainder of this section demonstrates how the composition in this case benefits from processes being independent compared to hierarchically structured systems.

Population lumping

The example contains homogeneous housings with uniform parameters. This is not unrealistic, given the uniformity of communities in suburban areas. Each housing is modeled as a process. The goal is to construct one DTMC as surrogate transition model for one housing in the community. By multiplying it with the Kronecker product, the probability of too many houses within the population switching simultaneously can be computed. Lumping can be applied between each two Kronecker multiplications to minimize the product chain to a counting abstraction.

The complexity of the aggregate DTMC of all households depends on the number of households and the granularity of the applied binning. This is similar to the size of the state space being the product over all register domains in the previous examples. The size of the full product chain here is n^k with n bins per household and k households. Notably, each bin has to be accounted for twice: once for *on* and once for *off* mode as shown in figure 7.6. In order to arrive at a tractable Markov chain, it is reasonable to select a binning according to the number of households such that the full product chain remains tractable. Assume the following symbolic DTMC $\mathcal{D}_1$ for one terrace housing as given.

The states are labeled *number status* again a described above in paragraph "Deterministic execution". Probability p_i is the probability that the temperature in a house remains in its current bin i for one time step and $1 - p_i$ is the probability that it progresses to the next temperature bin. The matrix is intentionally designed simple with only two bins and a sparse matrix to demonstrate lumpability.

$\downarrow$ from/to $\rightarrow$	1 on	2 on	1 off	2 off
1 on	p_1	$1 - p_1$		
2 on		p_2	$1 - p_2$	
1 off			p_3	$1 - p_3$
2 off	$1 - p_4$			p_4

Table 7.2: Example symbolic DTMC for one surrogate housing, $\mathcal{D}_1$

Consider the Markov chain to be irreducible and labeled $\mathcal{D}_1$. With the houses being mutually independent and executing equation 7.1 in parallel, maximal parallel execution semantics apply. In this case, the $\otimes$ operator specified in algorithm 16 coincides with the Kronecker product.

The product Markov chain of two houses with uniform parameters is the Kronecker product of two Markov chains $\mathcal{D}_1$. It calculates to $\mathcal{D}_1 \otimes \mathcal{D}_1$ and is labeled $\mathcal{D}_2$ — the index in $\mathcal{D}_i$ refers to the number of households — shown in table 7.3 on the next page. Empty quadrants are omitted according to the scheme shown in figure 7.4, in which the black cells represent the omitted zero values.

Table 7.4: Omission scheme for lumping the DTMC in table 7.3

Lumping is conducted as described in chapter 5 to reduce the DTMC shown in table 7.3 to the DTMC shown in table 7.5. The state lumping follows the schematics shown in figure 7.7 which describes the equivalence classes. It also shows the symmetry of the equivalence classes in the state space: The states mirrored at the diagonal or pairwise bisimilar. States $\langle 1\text{on}, 2\text{on} \rangle$ and $\langle 2\text{on}, 1\text{on} \rangle$ become state $\langle \mathbb{1}\text{on}, 2\text{on} \rangle$. The other equivalence classes are labeled analogously.

↓ first quarter row

from/to →	first quarter column				second quarter column			
	⟨1on,1on⟩	⟨1on,2on⟩	⟨1on,1off⟩	⟨1on,2off⟩	⟨2on,1on⟩	⟨2on,2on⟩	⟨2on,1off⟩	⟨2on,2off⟩
⟨1on,1on⟩	p_1^2	$p_1\cdot(1-p_1)$			$p_1\cdot(1-p_1)$	$(1-p_1)^2$		
⟨1on,2on⟩		$p_1\cdot p_2$	$p_1\cdot(1-p_2)$			$(1-p_1)\cdot p_2$	$(1-p_1)\cdot(1-p_2)$	
⟨1on,1off⟩			$p_1\cdot p_3$	$p_1\cdot(1-p_3)$			$(1-p_1)\cdot p_3$	$(1-p_1)\cdot(1-p_3)$
⟨1on,2off⟩	$p_1\cdot(1-p_4)$			$p_1\cdot p_4$	$(1-p_1)\cdot(1-p_4)$			$(1-p_1)\cdot p_4$

↓ second quarter row

from/to →	second quarter column				third quarter column			
	⟨2on,1on⟩	⟨2on,2on⟩	⟨2on,1off⟩	⟨2on,2off⟩	⟨1off,1on⟩	⟨1off,2on⟩	⟨1off,1off⟩	⟨1off,2off⟩
⟨2on,1on⟩	$p_2\cdot p_1$	$p_2\cdot(1-p_1)$			$(1-p_2)\cdot p_1$	$(1-p_2)\cdot(1-p_1)$		
⟨2on,2on⟩		p_2^2	$(1-p_2)\cdot p_2$			$(1-p_2)\cdot p_2$	$(1-p_2)^2$	
⟨2on,1off⟩			$p_2\cdot p_3$	$p_2\cdot(1-p_3)$			$(1-p_2)\cdot p_3$	$(1-p_2)\cdot(1-p_3)$
⟨2on,2off⟩	$p_2\cdot(1-p_4)$			$p_2\cdot p_4$	$(1-p_2)\cdot(1-p_4)$			$(1-p_2)\cdot p_4$

↓ third quarter row

from/to →	third quarter column				fourth quarter column			
	⟨1off,1on⟩	⟨1off,2on⟩	⟨1off,1off⟩	⟨1off,2off⟩	⟨2off,1on⟩	⟨2off,2on⟩	⟨2off,1off⟩	⟨2off,2off⟩
⟨1off,1on⟩	$p_3\cdot p_1$	$p_3\cdot(1-p_1)$			$(1-p_3)\cdot p_1$	$(1-p_3)\cdot(1-p_1)$		
⟨1off,2on⟩		$p_3\cdot p_2$	$p_3\cdot(1-p_2)$			$(1-p_3)\cdot p_2$	$(1-p_3)\cdot(1-p_2)$	
⟨1off,1off⟩			p_3^2	$p_3\cdot(1-p_3)$			$(1-p_3)\cdot p_3$	$(1-p_3)^2$
⟨1off,2off⟩	$p_3\cdot(1-p_4)$			$p_3\cdot p_4$	$(1-p_3)\cdot(1-p_4)$			$(1-p_3)\cdot p_4$

↓ fourth quarter row

from/to →	fourth quarter column				first quarter column			
	⟨2off,1on⟩	⟨2off,2on⟩	⟨2off,1off⟩	⟨2off,2off⟩	⟨1on,1on⟩	⟨1on,2on⟩	⟨1on,1off⟩	⟨1on,2off⟩
⟨2off,1on⟩	$p_4\cdot p_1$	$p_4\cdot(1-p_1)$			$(1-p_4)\cdot p_1$	$(1-p_4)\cdot(1-p_1)$		
⟨2off,2on⟩		$p_4\cdot p_2$	$p_4\cdot(1-p_2)$			$(1-p_4)\cdot p_2$	$(1-p_4)\cdot(1-p_2)$	
⟨2off,1off⟩			$p_4\cdot p_3$	$p_4\cdot(1-p_3)$			$(1-p_4)\cdot p_3$	$(1-p_4)\cdot(1-p_3)$
⟨2off,2off⟩	$p_4\cdot(1-p_4)$			p_4^2	$(1-p_4)^2$			$(1-p_4)\cdot p_4$

Table 7.3: Example TCL DTMC composition $\mathcal{D}_2$, 16 states, 64 transitions

$$\langle 1\text{on}, 1\text{on}\rangle \quad\quad \langle 1\text{on}, 2\text{on}\rangle \longrightarrow \langle 1\text{on}, 1\text{off}\rangle \quad\quad \langle 1\text{on}, 2\text{off}\rangle$$

$$\langle 2\text{on}, 1\text{on}\rangle \quad\quad \langle 2\text{on}, 2\text{on}\rangle \quad\quad \langle 2\text{on}, 1\text{off}\rangle \quad\quad \langle 2\text{on}, 2\text{off}\rangle$$

$$\langle 1\text{off}, 1\text{on}\rangle \quad\quad \langle 1\text{off}, 2\text{on}\rangle \quad\quad \langle 1\text{off}, 1\text{off}\rangle \quad\quad \langle 1\text{off}, 2\text{off}\rangle$$

$$\langle 2\text{off}, 1\text{on}\rangle \quad\quad \langle 2\text{off}, 2\text{on}\rangle \longleftarrow \langle 2\text{off}, 1\text{off}\rangle \quad\quad \langle 2\text{off}, 2\text{off}\rangle$$
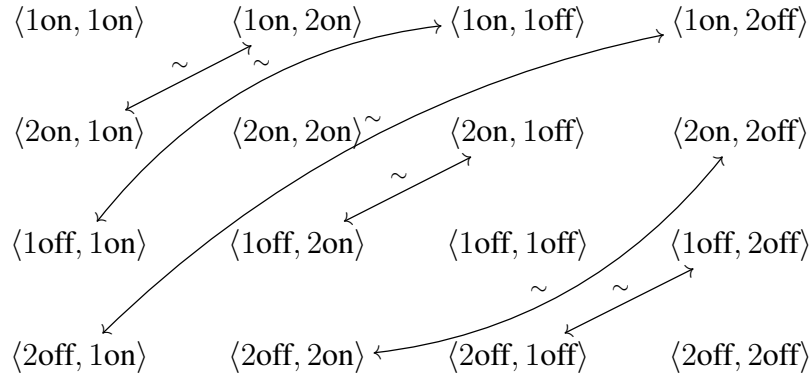
Figure 7.7: Lumping scheme

$\downarrow$ from/to $\rightarrow$	$\langle 1\text{on},1\text{on}\rangle$	$\langle 1\text{on},2\text{on}\rangle$	$\langle 1\text{on},1\text{off}\rangle$	$\langle 1\text{on},2\text{off}\rangle$	$\langle 2\text{on},2\text{on}\rangle$	$\langle 2\text{on},1\text{off}\rangle$	$\langle 2\text{on},2\text{off}\rangle$
$\langle 1\text{on},1\text{on}\rangle$	p_1^2	$2 \cdot p_1 \cdot (1 - p_1)$			$(1-p_1)^2$		
$\langle 1\text{on},2\text{on}\rangle$		$p_1 \cdot p_2$	$p_1 \cdot (1-p_2)$		$(1-p_1)\cdot p_2$	$(1-p_1)\cdot(1-p_2)$	
$\langle 1\text{on},1\text{off}\rangle$			$p_1 \cdot p_3$	$p_1 \cdot (1-p_3)$		$(1-p_1)\cdot p_3$	$(1-p_1)\cdot(1-p_3)$
$\langle 1\text{on},2\text{off}\rangle$	$p_1 \cdot (1-p_4)$	$(1-p_1)\cdot(1-p_4)$		$p_1 \cdot p_4$			$(1-p_1)\cdot p_4$

$\downarrow$ from/to $\rightarrow$	$\langle 1\text{on},2\text{on}\rangle$	$\langle 1\text{on},1\text{off}\rangle$	$\langle 1\text{on},2\text{off}\rangle$	$\langle 2\text{on},2\text{on}\rangle$	$\langle 2\text{on},1\text{off}\rangle$	$\langle 2\text{on},2\text{off}\rangle$	$\langle 1\text{off},1\text{off}\rangle$	$\langle 1\text{off},2\text{off}\rangle$
$\langle 2\text{on},2\text{on}\rangle$				p_2^2	$2\cdot(1-p_2)\cdot p_2$		$(1-p_2)^2$	
$\langle 2\text{on},1\text{off}\rangle$		$p_2 \cdot p_3$	$p - 2 \cdot (1-p_3)$			$(1-p_2)\cdot(1-p_3)$	$(1-p_2)\cdot p_3$	
$\langle 2\text{on},2\text{off}\rangle$	$p_1 \cdot (1-p_4)$	$(1-p_1)\cdot(1-p_4)$				$p_2 \cdot p_4$		$(1-p_2)\cdot p_4$

$\downarrow$ from/to $\rightarrow$	$\langle 1\text{on},1\text{on}\rangle$	$\langle 1\text{on},1\text{off}\rangle$	$\langle 1\text{on},2\text{off}\rangle$	$\langle 1\text{off},1\text{off}\rangle$	$\langle 1\text{off},2\text{off}\rangle$	$\langle 2\text{off},2\text{off}\rangle$
$\langle 1\text{off},1\text{off}\rangle$				p_3^2	$2\cdot(1-p_3)\cdot p_3$	$(1-p_3)^2$
$\langle 1\text{off},2\text{off}\rangle$		$p_3 \cdot (1-p_4)$	$(1-p_3)\cdot(1-p_4)$		$p_3 \cdot p_4$	$(1-p_3)\cdot p_4$
$\langle 2\text{off},2\text{off}\rangle$	$(1-p_4)^2$		$2\cdot(1-p_4)\cdot p_4$			p_4^2

Table 7.5: Lumped DTMC $\mathcal{D}_2'$, ten states, 36 transitions

This process can be repeated for k uniform households, that is, their respective transition models, until $\mathcal{D}_k'$ is composedly constructed.

Computing the complexity with enumerative combinatorics

Enumerative combinatorics provide the means to compute the number of states the lumped aggregate DTMC comprises. The state space explosion without lumping draws a state space according to *variation with repetition*. Therefore, there are $|\mathcal{S}| = n^k$ states when considering k houses and n bins. The successive lumping arrives at a state space of $|\mathcal{S}'| = \left(\binom{n}{k}\right) = \binom{n+k-1}{k} = \frac{(n+k-1)!}{(n-1)!\cdot k!}$ — the *multiset (rising) binomial coefficient* [Feller, 1968] — by *combination with repetition*. Figures 7.8(a) and 7.8(b) compare both state space explosions when adding more uniform households on the x-axis. They show that lumping *dampens* the explosion tremendously. The largest DTMC before the final lumping step in compositional lumping in this context is $\left(\binom{n}{k-1}\right) \cdot n$. Figure 7.8(a) compares the *initial* explosions up to ten households, while figure 7.8(b) computes the scalability for up to 100

households. The figures demonstrate that instead of the exponential state space explosion depicted in the red graphs, the size of the DTMC increases almost linearly with lumping, depicted in the blue graphs.
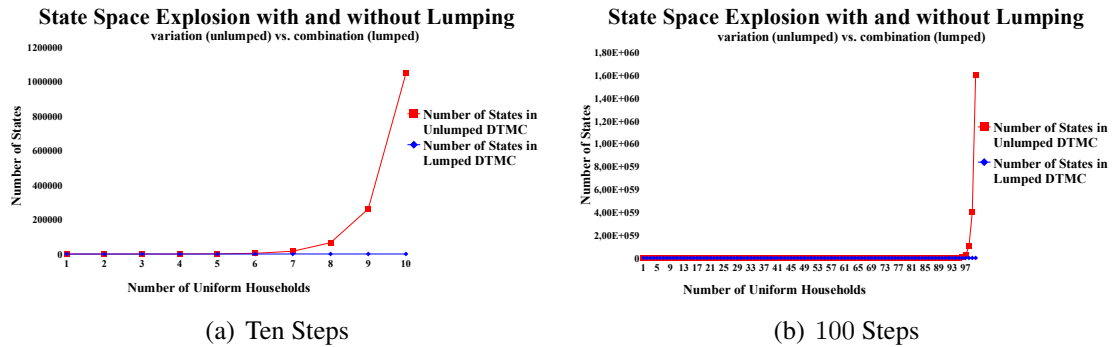

(a) Ten Steps

(b) 100 Steps

Figure 7.8: Dampening the state space explosion

Compared to unlumped multiplication, the graph under application of lumping almost coincides with the x-axis. Both complexities are computed with *enumerative combinatorics*, that is, variation and combination with repetition. The tractability of the DTMC depends on the available computing power. Even with lumping and perfectly homogeneous households, $\mathcal{S}'$ contains $176,851$ states for 100 housings and the proposed binning. Yet, compared to approximately $1.61 \cdot 10^{60}$ states, sequential composition and lumping are obviously preferable.

Control destroys bisimilarity

The sequential application of composition and lumping hinges on the mutual independence of the processes. *Control strategies* can prioritize housings to distribute limited resources, for instance when a limited amount of energy faces more demand than it can satisfy. In that case, processes loose their independence. The demand by one prioritized process can delay the satisfaction of another process.

For instance, assume that in the above example of $\mathcal{D}_2$ in table 7.3 one house constantly has a higher priority than the other one. Further, assume that the power grid cannot tolerate both thermostats switching simultaneously from on to off or vice versa. In case both thermostats desire to switch, the thermostat with the lower priority must wait exactly one time step. This adds two novel states to the system and replaces transitions accordingly as shown in table 7.6.

↓ from/to →	$\langle 2\text{on}, 2\text{on}\rangle$	$\langle 1\text{off}, 2\text{on}\rangle$	$\langle 2\text{on}, 1\text{off}\rangle$	$\langle 1\text{off}, 3\text{on}\rangle$
$\langle 2\text{on}, 2\text{on}\rangle$	p_2^2	$(1 - p_2) \cdot p_2$	$p_2 \cdot (1 - p_2)$	$(1 - p_2)^2$

↓ from/to →	$\langle 2\text{off}, 2\text{off}\rangle$	$\langle 1\text{on}, 2\text{off}\rangle$	$\langle 2\text{off}, 1\text{on}\rangle$	$\langle 1\text{on}, 3\text{off}\rangle$
$\langle 2\text{off}, 2\text{off}\rangle$	p_4^2	$(1 - p_4) \cdot p_4$	$p_4 \cdot (1 - p_4)$	$(1 - p_4)^2$

↓ from/to →	$\langle 1\text{off}, 1\text{off}\rangle$	$\langle 2\text{off}, 1\text{off}\rangle$	$\langle 1\text{on}, 1\text{on}\rangle$	$\langle 2\text{on}, 1\text{on}\rangle$
$\langle 1\text{off}, 3\text{on}\rangle$	p_3	$1 - p_3$		
$\langle 1\text{on}, 3\text{off}\rangle$			p_1	$1 - p_1$

Table 7.6: Prioritized TCL DTMC

In case the transition probabilities are not equal — $p_1 \neq p_2 \wedge p_1 \neq p_3 \wedge p_1 \neq p_4 \wedge p_2 \neq p_3 \wedge p_2 \neq p_4 \wedge p_3 \neq p_4$ — the DTMC becomes irreducible. For instance, the states $\langle 1\text{on}, 2\text{on}\rangle$ and $\langle 2\text{on}, 1\text{on}\rangle$ are then no longer probabilistic bisimilar as their outgoing transition probabilities would not coincide anymore as required by definition 5.1. Although the processes do not propagate values to one another thus excluding fault propagation, they depend on each other by sharing a mutual resource. When that resource is controlled, bisimilarity can be destroyed.

This paragraph demonstrated how sequential composition and lumping can be executed and pointed out that the absence of fault propagation does not necessarily imply independence of the processes. The example introduced control to destroy bisimulation among non-communicating processes. Next, a small numerical example computes the probability for a small community to suffer from a black out.

Sequential interleaving application of the $\otimes$ operator and lumping

Consider a set of 1000 households. For the sake of argument we assume the coarsest possible binning, yielding one bin for *on* and one for *off* mode. As discussed before, acquiring precise probabilities is not in the scope of this thesis. Therefore, we assume the following values as being provided. The probability to remain in the *on* bin is 0.9 and the probability to remain in the *off* bin is 0.8.

The full product chain without lumping contains $|\mathcal{S}| = 2^{1000} = 1,0715 \cdot 10^{301}$ states. When lumping is applied after each composition — which is a *counting abstraction* [Fu et al., 2002, p.195] —, the resulting DTMC contains only $|\mathcal{S}'| = 1001$ states, one state in which all are *off* one in which only one is *on* and so on until one state in which all 1000 are *on*. Its computation took about 50 minutes on a Intel(R) Core(TM) i5-3317U CPU at 1.7 GHz equipped with 8GB DDR3 SODIMM with MatLab. The source code is provided in appendix A.5.3. A graphical representation of the lumped product DTMC is shown in figure 7.9.
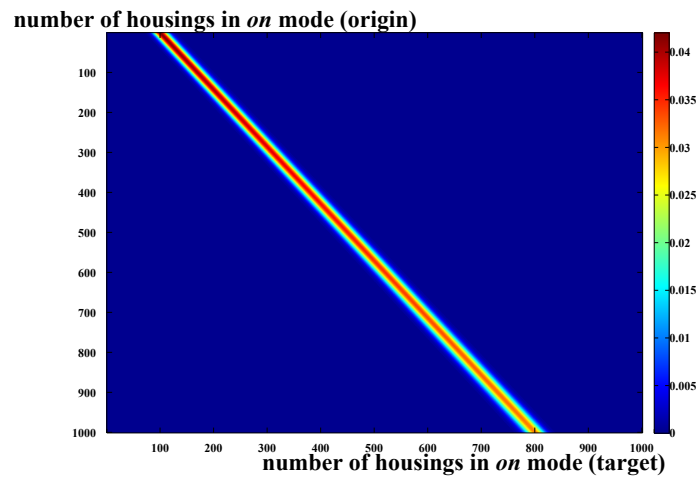
Figure 7.9: 1000 housings TCL power grid

Notably, there are no zero-probability transitions. The transitions in the *blue* areas are just very close to zero. The figure shows in the top row, in which all housings are *off*, a steep maximum at 100 housings simultaneously switching *on*. The bottom row in which all housings are *on* shows a shallower distribution with the maximum at 800 housings, indicating that about 200 housings simultaneously switch *off*.

With each housing being added, the matrix grows. Hence, each further addition takes longer than the previous one. The graph in figure 7.10 shows how the computation time of adding further housings increases with each housing.
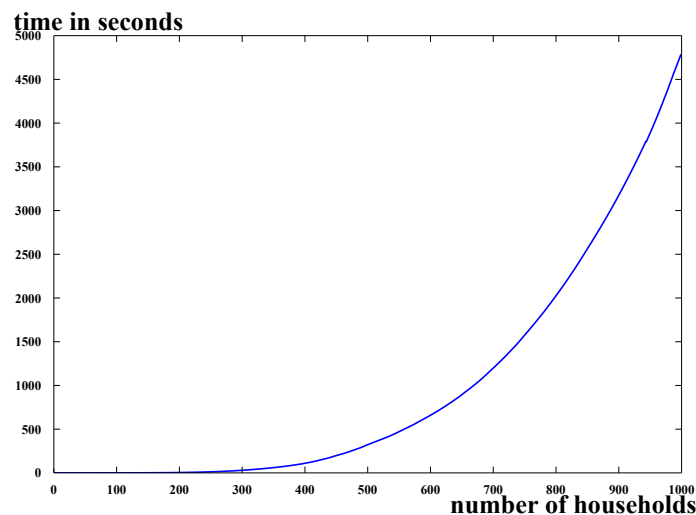
Figure 7.10: Time consumption to compute 1000 housings TCL power grid

Compared to decomposing hierarchical systems and otherwise mutually depending processes, composing mutually independent processes is rather simple. With the households being homogeneous, one surrogate DTMC can be multiplied with the Kronecker product over and over again with the resulting matrix being lumped after each iteration. The example in this section used the coarsest possible matrix — in which the θ domain of the temperature is not partitioned — to prove a point: Writing a script to compose independent processes can be as trivial as in this case. Then, generating matrices containing

thousands of states automatically is just a matter of time. The DTMC shown in figure 7.9 was generated in less than an hour on a tablet PC with the specifications given above. Contrary, constructing hierarchically structured systems is not as easy. The DTMC of the BASS example in the previous chapter contained only 648 states in its unreduced version and 324 states in its reduced state space. Its computation by hand took two weeks. Similarly, the DTMC of the example in the following section contains only 144 states and also took two weeks to compute by hand. Its computation is even more intricate than the computation of the BASS example although it contains less processes and also less states.

The number of states is not a good indicator to reason about scalability of the approach. Instead, the effort that is required to construct a DTMC to compute the desired measure can be used as an indicator. In the BASS example there were seven scheduler probabilities $\mathfrak{s}_1$ to $\mathfrak{s}_7$ and two fault probabilities p and q. For each of the 648 states there were hence 14 possible outcomes of an execution step, accumulating to 9072 possible outcomes overall in the unreduced matrix. The example in this had to regard merely two events for two states, accumulating to just four possible outcomes overall. The example in the following section regards two switches, two faults, ten scheduling decisions and 324 states, accumulating to 12960 possible outcomes overall in the unreduced matrix. Discussing scalability is not answered by the size of the state space, but by how complex the transition model is and how far the combination of decomposition and lumping allows to dampen the state space explosion. While this thesis provides the basic concepts that are necessary to reason about decomposing hierarchical systems, one promising goal for the future is hence an automatized method — including automatized slicing — like it is available for independent processes.

Safety in the context of power grid stability

From the consumer point of view, *safety* — the desired mode of operation — concerns the temperature. The current temperature should not deviate too much from the set temperature. From the supplier point of view, the system is safe when no voltage peaks occur. Such peaks occur when overall too many housings switch from on to off or vice versa. For instance, when 500 housings switch on while in the same step 500 housings switch off, then overall nothing changes. Redundancy here is the amount of houses that the power grid can cope with to simultaneously switch overall. Consider a community of 1000 housings. When all housings are *off*, one can expect about 100 housings to turn on again. On the contrary, when all housings are *on*, about 200 are expected to switch *off*. Yet, in the limit, the chances that all houses are either *on* or *off* are rather slim. To compute the chances of a black out we must compute the chances of *too many* housings — which means more than the system can cope with — simultaneously switch overall weighted with the stationary distribution. The goal is to compute this measure for all possible number of housing sufficing to cause a blackout to determine, which number provides which reliability.

The risk of black out – limiting window reliability

The probability that the system blacks out is the accumulated transition probability of too many houses switching on or off simultaneously. For instance, if the system blacks out with 1000 simultaneous houses switching, the probability for a black out computes as $pr(\overrightarrow{0,1000}) \cdot pr_\Omega(\langle 0 \rangle) + pr(\overrightarrow{1000,0}) \cdot pr_\Omega(\langle 1000 \rangle)$. The index here refers to the number of simultaneous switches necessary to cause a black out. When the system breaks down for

even 999 simultaneous switches, the probability for black out computes as $pr(\overrightarrow{0, 1000}) \cdot pr_\Omega(\langle 0 \rangle) + pr(\overrightarrow{1000, 0}) \cdot pr_\Omega(\langle 1000 \rangle) + pr(\overrightarrow{0, 999}) \cdot pr_\Omega(\langle 0 \rangle) + pr(\overrightarrow{999, 0}) \cdot pr_\Omega(\langle 999 \rangle) + pr(\overrightarrow{1, 1000}) \cdot pr_\Omega(\langle 1 \rangle) + pr(\overrightarrow{1000, 1}) \cdot pr_\Omega(\langle 1000 \rangle)$ and so forth. Figure 7.11(a) shows the stationary distribution that is required to compute the probability to crash. Figure 7.11(b) shows the probability to crash according to the required number of simultaneous switches that are required for the system to black out.

(a) Stationary distribution

(b) Risk per time step to black out depending on the number of simultaneously switching households
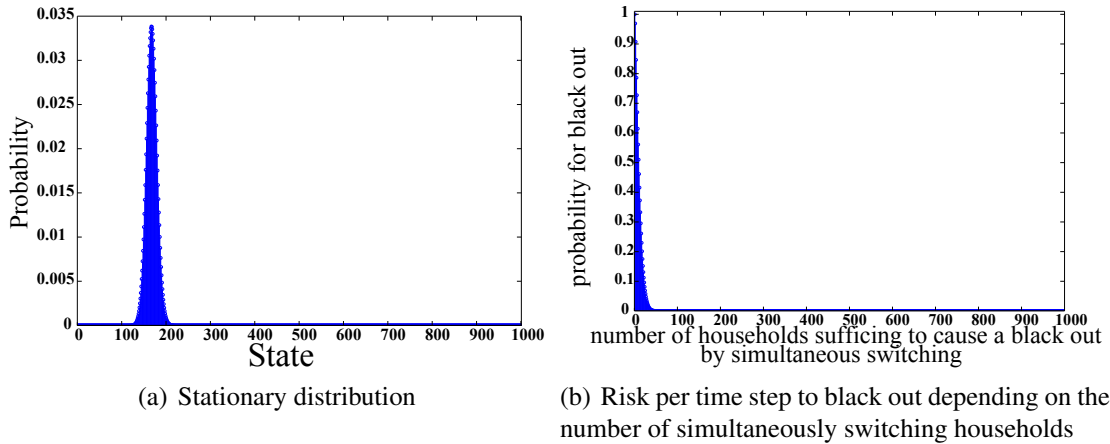
Figure 7.11: Determining the risk to crash

In this scenario we are not interested in the probability that a system converges timely as specified via LWA, but in the probability that closure is not violated within a given time window. The *limiting window reliability* in this context is similar to LWA, a probability distribution on first stopping times. In contrast to LWA it is a probability on stopping times of taking *a wrong* transition that violates closure while LWA measured the probability of taking *a right* transition to achieve convergence. The limiting window reliability, which is the chance to *survive* a given time window w without a black out, is simply computed with $(1 - pr_i)^w$ with respect to the critical accumulated number of households i that synchronously switch to evoke a black out.

With the limiting window reliability distribution, the ongoing risk of eventually suffering from a black out can be computed. Figure 7.12 shows how the probability for each *safety predicate* — that is, that either 1 house suffices to cause a black out, or 2, or 3 . . . — converges to 1 over time. The axis showing the *number of households sufficing to cause a blackout* is cropped at 100 but extends to 1000. With fewer houses required to cause a black out, the probability for a black out increases at a faster pace. The figure shows that more than about 60 houses are required in the predicate to pose a thread for the community to survive the first 100 time steps from the limit onwards. The algorithm to compute limiting window reliability for this case is provided in appendix A.5.3.
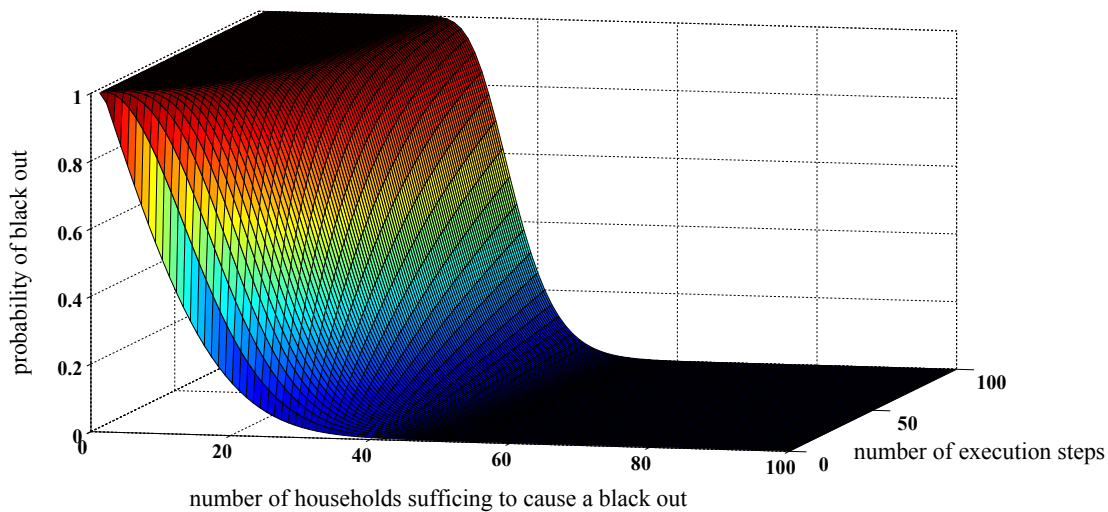
Figure 7.12: Limiting window reliability over 100 time steps

Figure 7.13 shows the same plot with the reliability being encoded via color for a larger time scale. This perspective nicely shows that i) the demarcation between unreliable (dark red) and reliable (dark blue) is very sharp (white) and ii) providing a safety threshold of even less than 100 houses can already suffice to provide for a high reliability for a time window size of at least 10,000 computation steps. If the available energy buffer — probably continuously realized via a battery — can cover for about 100 housings and furthermore superfluous excess energy is ventable, the system is quite reliable. Notably, the critical number of households required for simultaneous switching to cause a black out that the graph converges to in the limit is not a value about 50 as figure 7.13 might insinuate, but the total number or housings. What seems like counting to infinity twice — the limiting window reliability starts in the limit and then runs to the limit again — provides an important discussion. Similar to *limiting reliability* as discussed by Trivedi [Trivedi, 2002, p.321], the limiting window reliability for a limiting window is zero, too. In the limit, the red area extends to 1000 households sufficing to cause a black out.
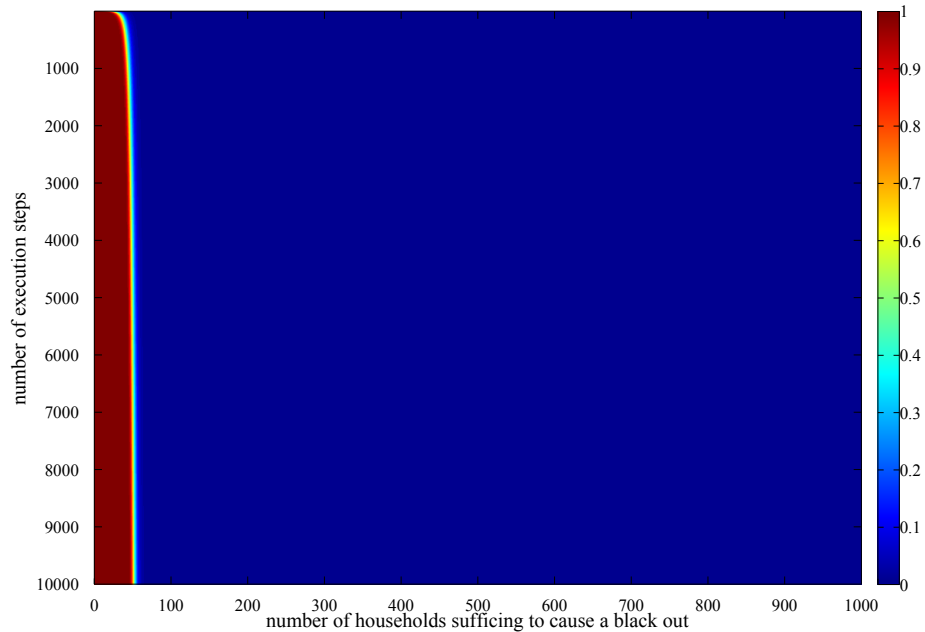
Figure 7.13: Limiting window reliability over 10,000 time steps

The result

The example demonstrated how a transition model and safety specifications can be derived from a real world system and how they can be utilized in the context of this thesis. With an initial probability distribution, the risk of eventually taking a transition in which too many households simultaneously switch in the same direction can be computed. Contrary to LWA that computes the probability that the system reaches the legal states within a time window, the *window reliability* computes the probability that a system leaves the legal states in a time window. Contrary to the *limiting reliability* [Trivedi, 2002, p.321] — with a probabilistic fault model the reliability of a system is zero in the limit — , it is reasonable in this context to compute the limiting *window* reliability. Consider the system to be initially supported by a vent and a buffer to compensate for voltage peaks until it converges sufficiently close to its stationary distribution. From thereon, the limiting window reliability determines the probability with which the system *survives* (cf. Appendix A.4.8) a desired time window by adding up all the relevant transition probabilities over that time.

Concluding power grids

This section provided a practical case study to discuss important aspects. It highlighted

- that determining probabilistic inputs is crucial to acquire realistic results with the presented methods and concepts,

- that a DTMC and a safety predicate can be derived from some real world systems for which fault tolerance properties shall be determined,

- that absence of fault propagation does not automatically imply process independence and thus bisimilarity, and

- that the notion of limiting or instantaneous window properties can be easily adapted to suit a desired context.

It furthermore demonstrated how synthesizing a transition model benefits from the processes being independent. The constructed DTMC accounts for thousand processes and was generated in less than an hour. On the contrary, the transition models of hierarchic systems are not constructed that easy. Yet, this thesis provides the concepts and methods to apply decomposition and lumping in their context.

7.2 A semi-hierarchical, semi-parallel stochastic sensor network

The BASS example in chapter 6 had to take several restrictions to allow for a comprehensive description.

- The influence allowed only for deterministic and probabilistic influence so far.

- With serial execution semantics, parallel execution with regards to subsystems and scheduling could not be discussed.

- The processes were organized strictly hierarchically.

- Slicing in multiple processes was addressed only informally.

The setting in this section is selected with the intention to demonstrate that the methods and concepts that have been described for a restrictive setting can also cope with more complex settings.

The setup

A greater space like a desert or vineyard [Burrell et al., 2004] is covered evenly with such small local networks as depicted in figure 7.14. Each sensor mote — modeled as a process — in such a local wireless sensor network (WSN) is supposed to measure either humidity or temperature, exclusively one of them at a time. A broadcast station radios the type of value that shall be stored.
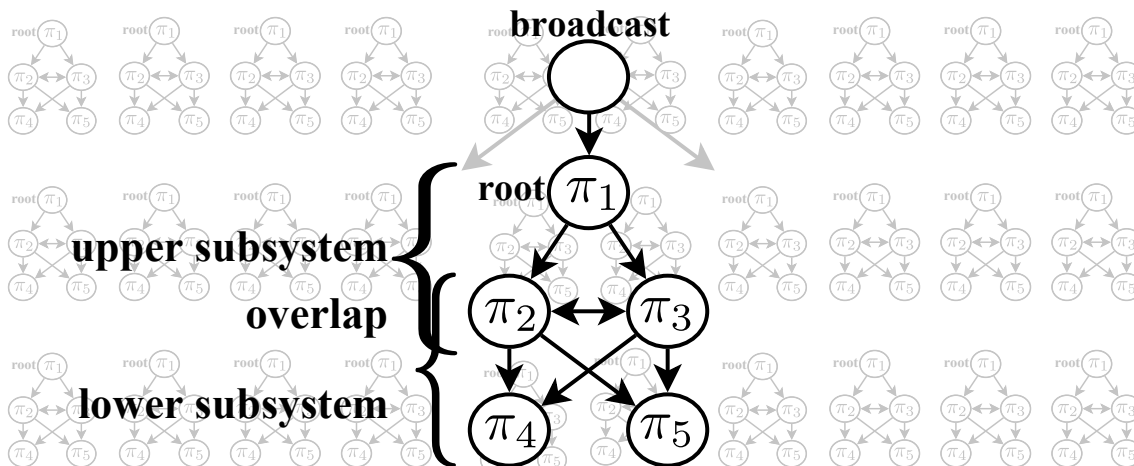
Figure 7.14: Small wireless sensor network

Since radio receivers are expensive and the lower sensor motes have bad reception, only the top sensor mote is equipped with a radio receiver. This example focuses on the evaluation of a single local WSN. We are interested in the transition model of one local WSN that, when required, can be composed as discussed in the previous section to account for a whole field.

Each process contains both sensors and enough memory to store measured data for the duration of the desired mission time. The sensors can only measure one kind of data per time step. The root process reads the broadcasted value and propagates it to all neighboring processes. The switch between measuring temperature or humidity is arbitrary and modeled probabilistically. Consider for instance an observer that wants to evaluate temperature and humidity of a region. That observer can switch the type of data to be recorded. With a probability pr_{switch} they change want *the other* measure to be recorded and with probability $1 - pr_{switch}$ they continue with the same value.

Process π_1 is the root process and reads only the probabilistic broadcast. The non-root processes behave similar to the BASS algorithm but without priorities. Processes π_2 and π_3 read from π_1 and from each other. Processes π_4 and π_5 read from both π_2 and π_3. A central scheduler demon that is not shown in the figure probabilistically selects two processes per time step to execute in parallel. Each computation step starts with a read phase in which the executing processes inquire which type of data is to be stored. Afterwards, they store the corresponding measure. Forcing the processes to maintain a strict sequence of reading and writing allows to exclude read-after-write hazards as discussed by Patterson and Hennessy [Hennessy and Patterson, 1996, Patterson and Hennessy, 2005]. For instance, assume that π_1 and π_2 execute. Without a strict sequence it is undetermined whether first π_1 updates according to the radio broadcast and also updates π_2 in the same step, or if first π_2 inquires the *old* status of π_1 before both processes write. With the strict sequence the latter constellation is considered.

We abbreviate temperature with 0 and humidity with 2. Contrary to the BASS example 2 is not a *fault* value. When a process cannot determine which the intended type is — that is, when there is no majority for one of the two types —, it stores nothing to save memory, and propagates 1 until it executes again. The value 1 coincides with the *don't know* value in the BASS example. The fault model forces a process to store 0 when it should store 2 and vice versa. In case a process is supposed to store 1 and is perturbed by a fault, the effect of the fault is undetermined. We pessimistically assume that it stores the currently inappropriate value that is not broadcasted at that time.

With five processes and two processes executing in parallel per time step, it requires at least five steps for every processes to have executed. Hence, after a switch on the broadcast, the system must execute at least three steps to propagate the new type of data to be stored. The system thus cannot continuously store the desired data in every process.

With $pr_{switch} \geq \frac{1}{3}$, meaning that a switch occurs in average every three or less time steps, it is unlikely that — even without transient faults — the consistency is very high, since the mean switching interval is lower than the minimal time required for convergence.

The goal

The goal is to determine the *consistency* of the measured data in a given probabilistic environment. A *set of data* contains the data stored by each process. That set is consistent if it coincides with the broadcasted type at that time. The system has not only to cope

with transient faults propagating through the system, but also with probabilistic switches. The system probabilistically converges to the currently broadcasted type of data and is thereby probabilistically self-stabilizing. In this case, the LWA is adapted to cover not only for one time step, but for each time step during the whole mission time. What is the probability that each process stores the desired type of data in each time step? Furthermore, we assume that all processes initially store 0 and 0 is broadcasted in the first time step. Thereby, we are interested in the instantaneous window availability with window size 1 for each time step.

The motivation

This example allows to highlight four important properties that have been discussed in the beginning of this section. The first point concerns resolving non-determinism. In the fault model it is undetermined which value the perturbed process stores when it is supposed to store 1. In the example, we pessimistically assume that the currently not broadcasted value is stored to resolve this non-determinism, thus computing the *lower boundary* of the probability that all processes store the currently broadcasted value. The same computation can be repeated with an optimistic assumption — that is, storing the currently broadcasted value — leading to the *upper boundary* of the probability that all processes store the currently broadcasted value. The upper and lower boundaries demarcate the corridor of possible execution traces. Notably, for the case of the non-deterministic decisions being *controllable*, interactive Markov chains can provide a suitable transition model [Hermanns, 2002] to replace DTMCs. Similar to reducing DTMCs, their bisimilar states can be reduced as well as for instance discussed for independent processes by Hermanns and Katoen [Hermanns and Katoen, 2009]. Replacing the transition model is discussed in chapter 8.

The second variation to the previous chapter are parallel execution semantics. In the BASS example, the DTMC was split and the adapted Kronecker product accounted for that either exclusively in the upper subsystem Π_1 or exclusively in the lower subsystem Π_2 one process executed. Contrary, parallel execution semantics as in section 7.1 are not challenging as the processes do not compete for the resource of execution. *Semi-parallel execution semantics* — that is, not *all* processes but more than *one* process can execute per time step — are a special challenge as a case distinction is necessary that is neither required for strictly serial execution semantics nor for parallel execution semantics. The present case study allows to address this challenging issue.

Third, the system contains a heterarchical subsystem with processes π_2 and π_3 being locally heterarchical. As discussed before, decomposing heterarchical (sub-)systems or subsystems is not possible offhand. Therefore, they are to be always kept together during the decomposition. With the discussion about overlapping sets on page 85 in mind, the challenge here will be to slice the system through the heterarchical set. The two heterarchical processes are the overlapping set.

This also allows to discuss the fourth alteration. Slicing through multiple processes was discussed only informally before. This example allows to show how slicing through multiple processes is not more complex than slicing in one process. The slicing in this example further demonstrates that the processes in subsystems — in this case π_4 and π_5 — need not even necessarily be connected.

The input parameters

The input parameters contain

1. fault probabilities,

2. switching probabilities and

3. scheduling probabilities.

We consider the fault probability of an executing process storing the wrong type of data to be $q = 0.01$ and the switching probability to be $pr_{switch} = 0.03$ for both switching directions, from 0 to 2 and vice versa. The numerical values[2] can be adapted as desired. The scheduler selects two processes randomly with a uniform probability distribution.

The safety predicate

The system is in a safe state — that is, the data set being recorded is consistent — when all processes record the value that is broadcasted at that time:

$$s_t \models \mathcal{P} \begin{cases} s_t = \langle 0, 0, 0, 0, 0 \rangle \wedge \text{broadcasted value is } 0 \\ s_t = \langle 2, 2, 2, 2, 2 \rangle \wedge \text{broadcasted value is } 2 \end{cases} \tag{7.3}$$

The quantification method can easily be adapted such that safety is also satisfied when not all, but only a subset of the processes stores the broadcasted type of value.

The state spaces

The first process can store either 0 or 2 and all other processes can derive 1 as well. Furthermore, the broadcasted value determines whether $s_t \models \mathcal{P}$ and must be accounted for as well. For instance, the system can be in state $s_t = \langle 0, 0, 0, 0, 0 \langle$ when 0 is broadcasted, thus satisfying $\mathcal{P}$, or it can be in the same state when 2 is broadcasted, thus not satisfying $\mathcal{P}$. The full product transition model hence contains $|\mathcal{S}| = 2 \cdot 2 \cdot 3^4 = 324$ states — that is, number of possibly broadcasted values times the number of possible values in π_1 times the number of possible states to the power of processes these are being stored in — as pictured in figure 7.15(a).

(a) Full state space   (b) Lumped state space
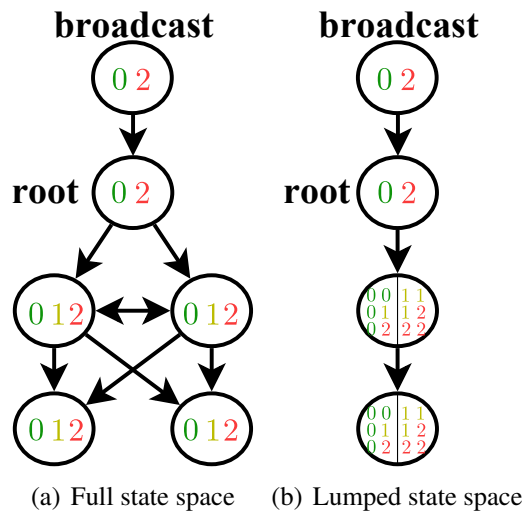
Figure 7.15: State space reduction

Coalescing of states results in a state space of $|\mathcal{S}'| = 2 \cdot 2 \cdot 6 \cdot 6 = 144$ states.

The decomposition

The system is sliced in π_2 and π_3 as discussed in chapter 6. The upper subsystem comprises processes π_1, π_2 and π_3. The lower subsystems contains processes π_4 and π_5. Contrary to the proceeding in the BASS example, the overlapping processes π_2 and π_3 are awarded to the upper subsystem during the uncoupling of the decomposition as described in paragraph "Uncoupling with $\otimes$" on page 89. The scheduler selects two processes. If the two processes were to deterministically execute both within the same subsystem, the decomposition could be carried out like for serial execution semantics. Here, two processes, one in each subsystem, can execute in parallel. Therefore, each case must be accounted with its own transition matrix. The first case is that both processes selected for execution belong to the upper subsystem. The second case is that both selected processes belong to the lower subsystem. The third case is that one process belongs to each of the two subsystems.

We label the sub-Markov chain for the upper subsystem $\mathcal{D}_1$ and $\mathcal{D}_2$ for the lower subsystem. A second index is added labeling the case if no process in the corresponding subsystem is selected $\mathcal{D}_{1,0}$, if one process is selected $\mathcal{D}_{1,1}$ or if both selected processes are within the subsystem $\mathcal{D}_{1,2}$ (analogously for $\mathcal{D}_2$). Figure 7.16 shows the decomposition schema.
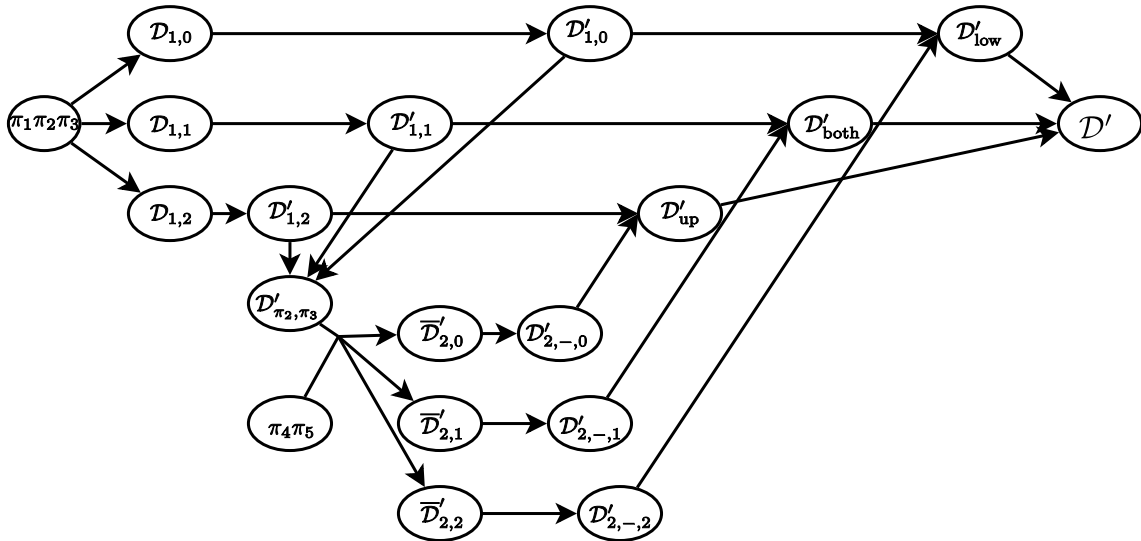
Figure 7.16: Decomposing the WSN transition system

Like in the BASS example, the upper subsystem — being hierarchically superior — is tackled first. A graphical representation of the sub-Markov chains is provided in appendix A.5.4. The transition matrices describe what can happen in one execution step with two processes executing simultaneously. The three probabilistic influences are switch, fault and scheduler selection. The latter comprises the events $\mathfrak{s}_{1,2}$, the probability that processes π_1 and π_2 are selected, $\mathfrak{s}_{1,3}$, $\mathfrak{s}_{1,4}$, $\mathfrak{s}_{1,5}$, $\mathfrak{s}_{2,3}$, $\mathfrak{s}_{2,4}$, $\mathfrak{s}_{2,5}$, $\mathfrak{s}_{3,4}$, $\mathfrak{s}_{3,5}$ and $\mathfrak{s}_{4,5}$. With uniformly distributed scheduling probabilities, each combination is likely to be selected with 0.1. The probability that exactly one process in the upper subsystem is selected is

hence $\mathfrak{s}_{1,4} + \mathfrak{s}_{1,5} + \mathfrak{s}_{2,4} + \mathfrak{s}_{2,5} + \mathfrak{s}_{3,4} + \mathfrak{s}_{3,5} = \mathfrak{s}_{both} = 0.6$. The probability that both selected processes belong to the upper subsystem is $\mathfrak{s}_{1,2} + \mathfrak{s}_{1,3} + \mathfrak{s}_{2,3} = 0.3 = \mathfrak{s}_{up}$. The probability that none of them is selected is $\mathfrak{s}_{4,5} = 0.1 = \mathfrak{s}_{low}$.

For each case, the transition matrix is constructed as described in section 2.4. Next, all three matrices $\mathcal{D}_{1,0}$, $\mathcal{D}_{1,1}$ and $\mathcal{D}_{1,2}$ are lumped to $\mathcal{D}'_{1,0}$, $\mathcal{D}'_{1,1}$ and $\mathcal{D}'_{1,2}$. These matrices are required again later. Afterwards, the three matrices $\mathcal{D}'_{1,0}$, $\mathcal{D}'_{1,1}$ and $\mathcal{D}'_{1,2}$ are added and $\mathcal{D}'_{\pi_2,\pi_3}$ is uncoupled.

Both π_4 and π_5 store

- 0 when reading $\langle 0, 0 \rangle$ or $\langle \mathbb{1} \rangle$, which is the lumped state of states $\langle 0, 1 \rangle$ and $\langle 1, 0 \rangle$,

- 1 when reading $1, 1$ or $\langle \mathbb{2} \rangle$ which is the lumped state of states $\langle 0, 2 \rangle$ and $\langle 2, 0 \rangle$, and

- 2 when reading $\langle 2, 2 \rangle$ or $\langle \mathbb{3} \rangle$ which is the lumped state of states $\langle 1, 2 \rangle$ and $\langle 2, 1 \rangle$.

A minor simplification

At this stage we exploit a minor simplification that is helpful when computing the *instantaneous* window availability instead of a limiting property. When computing a limiting property, the limiting probability that a certain input is propagated from superior to inferior subsystem does not change over time. It is the same for time step Ω as it is for time step $\Omega + 1$. For *instantaneous* properties on the other hand, the probability that a certain values is propagated *does* change, until reaching the stationary distribution in the limit. For a precise quantification it would be necessary to construct the transition model for the lower subsystem for each time step.

The important question is, how this simplification influences the result. In the beginning, the input vector (i.e. the initial distribution) differs maximal from the stationary distribution. With each time step it differs less. When the stationary distribution replaces the current distribution as input parameter, convergence is sped up. The next question is, how much the convergence is sped up. Provided that the initial distribution assigns the complete probability mass to state $\langle 0, 0, 0, 0, 0 \rangle$ and that 0 is broadcasted in the beginning, the probability that this status changes is less 0.08. With each additional computation step the computation error gets slimmer until becoming zero in the limit. The simplification is considered to be acceptable since i) the convergence to the stationary distribution is quick and the introduced error only relevant to the first few computation steps, ii) the error converges to zero itself, and iii) the gain for accepting this slight error is a great simplification in the computation. The lower sub-Markov chains needs not be computed (and lumped subsequently) for each time step.

Continuing the construction of $\mathcal{D}'$

Thus, matrices $\overline{\mathcal{D}}'_{2,0}$, $\overline{\mathcal{D}}'_{2,1}$ and $\overline{\mathcal{D}}'_{2,2}$ are constructed. In these matrices, the states where π_2 and π_3 are responsible for bisimilarities have been lumped but the states where π_4 and π_5 are responsible for bisimilarities have not, hence the overline notation. Processes π_2 and π_3 are uncoupled subsequently. Lumping further reduces the state space and $\mathcal{D}_{2,-,0}$, $\mathcal{D}_{2,-,1}$ and $\mathcal{D}_{2,-,2}$ are constructed. Finally, $\mathcal{D}'_{low} = \mathcal{D}'_{1,0} \otimes_K \mathcal{D}'_{2,-,2}$, $\mathcal{D}'_{both} = \mathcal{D}'_{1,1} \otimes_K \mathcal{D}'_{2,-,1}$ and $\mathcal{D}'_{up} = \mathcal{D}'_{1,2} \otimes_K \mathcal{D}'_{2,-,0}$ are computed by applying the Kronecker product. Here, the Kronecker product is applicable since two processes execute in parallel. As the cases of parallel executions have been distinguished from the beginning, the full DTMC $\mathcal{D}'$ is the accumulated effort of all three case DTMCs: $\mathcal{D}' = \mathcal{D}'_{low} + \mathcal{D}'_{both} + \mathcal{D}'_{low}$.

The result

Figure 7.17 shows the probability mass in states $\langle 0,0,0,0,0 \rangle$ when 0 is propagated (green line converging from above) and $\langle 2,2,2,2,2 \rangle$ when 2 is propagated (red line converging from below) for the first thousand time steps. It merely takes a little more than a hundred steps until both lines meet and the system *converged*. The numerical values at this time-step are $pr(s_{100} = \langle 0,0,0,0,0 \rangle \wedge \text{propagated value} = 0) = 0.4151$ and $pr(s_{100} = \langle 2,2,2,2,2 \rangle \wedge \text{propagated value} = 2) = 0.4033$. With equal switching an fault probabilities it was expected that both predicate satisfaction probabilities converge to the same value. With switching at 0.03 and a minimum of three computation steps for convergence and a fault probability of 0.01 it seems plausible that the consistency is about 0.82.
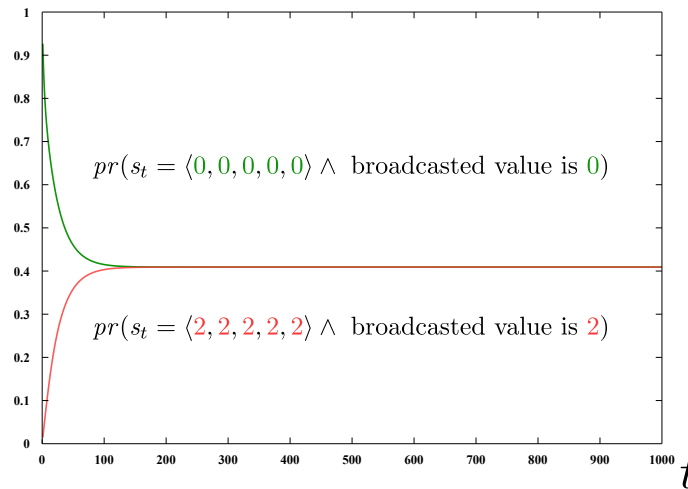
Figure 7.17: Result of the WSN example – converging consistency

The average consistency of the measured data is about 0.82 in the limit. The *convergence inertia* — which is the time spent for convergence due to both switching and recovery — is shown in figure 7.18. The term *inertia* is chosen as the system requires time to cope with switching and the effects of faults. The convergence inertia is the probability with regards to the current time step that the system is *between* the legal states.
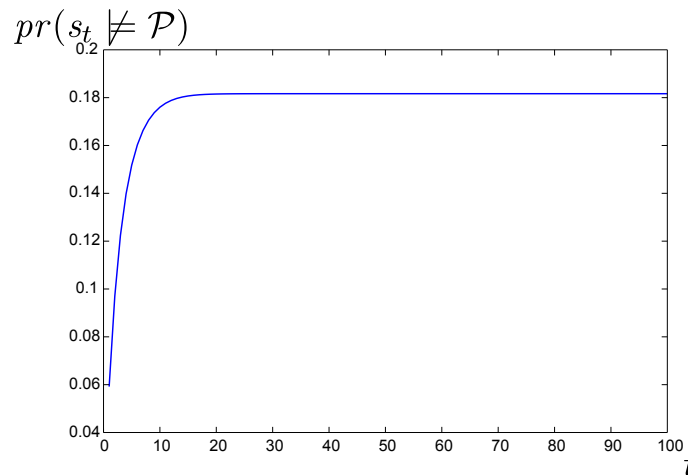
Figure 7.18: Result of the WSN example – convergence inertia

It converges to about 0.18 very fast which is why it is plotted only for the first 100 states.

Interpretation

While the LWA is a measure on stop times, this example demonstrates how such stop time can continuously be exploited by measuring the desired probability for *each* time step in contrast to just its first occurrence. The example further demonstrated how non-determinism, local heterarchies, semi-parallel execution semantics and slicing among multiple — even heterarchical lumpable — processes can be achieved. The notion of *switching* introduced a further challenge by demanding dynamic predicates and doubling the size of the state space. Furthermore, it extended pure *recovery liveness* during times of error to a more general *convergence inertia* that accounts for both switching as well as effects of faults. A system designer has now the opportunity to test various settings, alter the fault and switching probabilities as well as the topology, until a suitable system is found providing the desired consistency of the measured data.

7.3   Summarizing the case studies

The case studies demonstrated the practical benefit of the proposed methods and concepts and further pointed out challenges and limitations that could not be addressed before. The power grid example showed that it can be challenging to determine probabilistic influence and that composing sub-Markov chains of independent processes is a whole lot easier than composing hierarchical systems that are in the focus of this thesis. The second example required four extensions to the previous discussion, all of which could be coped with. Combining the lessons learned from both examples, decomposing *large* systems seems tractable when i) the whole system comprises mutually independent uniform components like in section 7.2, ii) the subsystems are structured hierarchically like in section 6.5, and heterarchies occur only locally and do not have to be further decomposed as in section 7.2.

8. Conclusion

The goal of this thesis was to develop methods and concepts for a probabilistic reasoning and quantification of the fault tolerance properties of systems that can recover from the effects of faults. Distinguishing systems into being hierarchical, semi-hierarchical and heterarchical allowed to demarcate the scope of this thesis: Hierarchical systems.

Chapter 1 introduced and motivated the goals of this thesis and chapter 2 presented system, fault and environment models as foundations to achieve these goals. The term *fault tolerance* is perceived differently in various contexts[1]. This thesis provided a general fault tolerance taxonomy in the context of this thesis in chapter 3. In this context, *limiting window availability* was proposed as a suitable measure in chapter 4. A technique to compute it, based on discrete time Markov chains, was presented. This technique derives from probabilistic model checking and suffers from *state-space explosion*. Lumping, an opportunity to cope with this issue, has been discussed in the context of computing LWA in chapter 5. Yet, to apply lumping, it was necessary to construct the full product DTMC. Decomposing the system to apply lumping locally has been addressed in probabilistic model checking for systems without fault propagation. The systems discussed in this thesis demand a different approach, yet, as they allow fault propagation as necessary evil to benefit from self-stabilization. A novel decomposition technique for hierarchical systems was developed to apply lumping locally on the considerably smaller sub-Markov chains of the subsystems. In this context, the Kronecker product, that has been applied for parallel and mutually independent processes, was successfully adapted to suit hierarchical systems and serial execution semantics. Chapter 6 explained the general decomposition method. To show the suitability of the concepts developed and to point out challenges and opportunities that could not be addressed before, chapter 7 provided two case studies. This chapter closes the thesis by contemplating about future directions.

The core contributions to the state of the art include:

- a sound and general fault tolerance taxonomy that extends to probabilism and non-determinism

- determining those aspects that are important regarding recovery dynamics of a system and probabilistic fault tolerance

---

[1]Appendix A.4 presents a variety of selected definitions of those terms constituting fault tolerance.

- window measures like instantaneous and limiting window availability and reliability to quantify the recovery dynamics

- providing a method based on DTMCs to compute these measures

- discussing lumping in this context

- introducing decomposition of hierarchically structured systems

- combining lumping and decomposition

- contrasting systems with mutually independent processes and hierarchical, semi-hierarchical and heterarchical (sub-)systems

- academical examples including TLA and BASS as well as practical examples like TCL and WSN

Future work

Limiting window availability is a specific measure. Instantaneous window availability was presented early to point out that this measure can easily be adapted to satisfy a different focus. Further variations concern safety to hold for more than one consecutive time step or to hold for a least number of computation steps within a time window. Further possible extensions to the discussion concern multiple root processes, approximate decomposition of heterarchic subsystems and to further investigate semi-parallel execution semantics.

The *trustworthiness* of the probabilistic data is a promising topic as well. As discussed in the introduction, the strength of the proposed approach lies in its precision. With probabilities for scheduling and faults being provided, the proposed analytic method allows to precisely compute the fault tolerance measures of a system in a probabilistic environment, thereby overcoming the limitations of sampling methods such as simulation and real world experiments. The Achilles heel of the approach is the quality of the assumed or estimated probabilities. The discussion about rare events pointed out how critical this precondition is.

The remaining directions proposed for future research aim at related work. The focus is on either exploiting methods and concepts from related domains in the context of this thesis, or, on the contrary, to disseminate the contributions of this thesis.

The concept of self-stabilization — as presented in section 3.2 — provides a rather strict corset to discuss hierarchically structured fault tolerant systems. Relaxing the convergence property to probabilistic self-stabilization [Devismes et al., 2008] allowed to account for probabilistic transient faults. It seems promising to reason about further relaxations. The work by Podelski, Wagner and Mitrohin [Mitrohin and Podelski, 2011, Podelski and Wagner, 2006] for instance discusses relaxations for hybrid systems that might be applicable in the context of this thesis.

The systems presented in this thesis were already minimal regarding the register domains. In reality, systems commonly feature much larger domains like integers or floats. Abstracting such domains to the relevant information is already discussed by Katoen et al. [Katoen et al., 2012]. Similarly, the continuous temperature domain in the TCL example was abstracted into bins. The relation between the granularity of *binning* and the precision of the resolving transition model is currently addressed by Soudjani and

Abate [Soudjani and Abate, 2013a]. When decomposition and lumping provide insufficient leverage, widening the bins might help in achieving the goal. Another similar angle is *approximate* lumping of *similar* states as for instance discussed by Mertsiotakis [Mertsiotakis, 1998]. The criteria for states to be *sufficiently* similar to qualify for lumping can be softened until the analysis of the system becomes tractable. It seems very promising to combine these relaxations and to determine a reasonable trade-off between approximate lumping and granularity of abstraction of parameter spaces. The challenge seems to be the computation of the precise error that is introduced by either relaxation in order to find the tractable solution with the smallest error.

This thesis focused solely on DTMCs as they naturally model the behavior of the systems under consideration. Yet, switching the transition model might be beneficial. For instance, Baier and Katoen [Baier and Katoen, 2008] as well as Mertsiotakis [Mertsiotakis, 1998] discuss a variety of transition models and their respective advantages. It might be worthwhile to discuss the developed methods and concepts for other transition models as well.

Considering other transition models consequently leads to the discussion about non-determinism. Assume not every single probability is known like in the WSN example. In that case, discrete time Markov decision processes or interactive Markov chains could replace the DTMCs. For instance, the papers by Zhang et al. [Zhang et al., 2010] and Fränzle et al. [Fränzle et al., 2011] discuss deterministic system dynamics under partially probabilistic, partially non-deterministic influence.

The task of probabilistic fault tolerance design is a problem of multi-objective optimization (cf. e.g. [Deb, 2001]). While the costs are to be minimized, the degree of fault tolerance is to be maximized. Section 3.5 motivated to focus on temporal redundancy to account for recovery liveness. Yet, systems might offer a huge variety of possibilities to allocate different resources in order to acquire different kinds of fault tolerance. On the other hand, systems might have more desirable properties than just fault tolerance, like performance or energy consumption. In this context, Andova et al. [Andova et al., 2003] provide a suitable extension to PCTL. Figure 8.1 depicts a system as a black box that has different kinds of redundancy — the *currencies* of fault tolerance — as input parameters and different kinds of fault tolerance as output parameters.
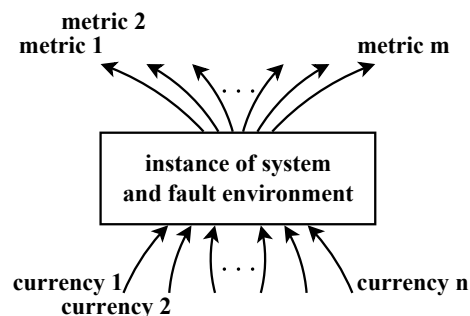

Figure 8.1: Black box fault tolerance design

Converting different kinds of redundancy into different kinds of fault tolerance or other quantifiable system properties is not always as easy as determining the efficiency of triple modular redundancy. With multiple inputs and multiple outputs confining the design space, a *counter-example guided abstraction refinement* (CEGAR) [Hermanns et al., 2008] of the system design seems in order. Once the internals

of the black box are specified — for instance as a symbolic DTMC —, the input parameters can be adjusted until the output parameters satisfy desired constraints.

Furthermore, the development of software to tackle hierarchical systems is desirable. Existing tools like PRISM already provide the opportunity to generate a transition model from a system definition. Adding to popular tools such as CADP or PRISM to support the automatic decomposition and lumping of hierarchical — and possibly even semi-hierarchical — systems and different execution semantics can be based on the concepts and methods developed in this thesis.

Bibliography

[IEE, 1988] (1988). IEEE Standard 982.1-1988. superseded by 982.1-2005 - IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.

[198, 1989] (1989). LOTOS - A formal description technique based on the temporal ordering of observational behaviour. Standard. Information Processing Systems, Open Systems Interconnection.

[IEE, 1990] (1990). IEEE Std 610.12-1990(R2002).

[ISO, 1999] (1999). ISO/IEC 14598-1: Information Technology - Software Product Evaluation - Part 1: General Overview.

[ISO, 2001] (2001). ISO/IEC 9126.

[Afek et al., 1997] Afek, Y., Kutten, S., and Yung, M. (1997). The Local Detection Paradigm and its Applications to Self-Stabilization. *Theoretical Computer Science*, 186:199 – 230.

[Alpern and Schneider, 1985] Alpern, B. and Schneider, F. B. (1985). Defining Liveness. Technical report, Ithaca, NY, USA.

[Andova et al., 2003] Andova, S., Hermanns, H., and Katoen, J.-P. (2003). Discrete-time Rewards Model-Checked. In Larsen, K. G. and Niebert, P., editors, *Formal Modeling and Analysis of Timed Systems (ForMATS)*, volume 2791 of *Lecture Notes in Computer Science*, pages 88–104. Springer Verlag.

[Armstrong, 2007] Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.

[Arora and Kulkarni, 1998a] Arora, A. and Kulkarni, S. S. (1998a). Designing Masking Fault-Tolerance via Nonmasking Fault-Tolerance. *IEEE Transactions on Software Engineering*, 24(6):435 – 450.

[Arora and Kulkarni, 1998b] Arora, A. and Kulkarni, S. S. (1998b). Detectors and Correctors: A Theory of Fault-Tolerance Components. In *International Conference on Distributed Computing Systems*, pages 436 – 443.

[Arora and Nesterenko, 2004] Arora, A. and Nesterenko, M. (2004). Unifying Stabilization and Termination in Message Passing Systems. *Distributed Computing*.

[Avižienis et al., 2001] Avižienis, A., Laprie, J.-C., and Randell, B. (2001). Fundamental Concepts of Dependability. pages 7 – 12.

[Avižienis et al., 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11 – 33.

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

[Baruah et al., 2012] Baruah, S. K., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., and Stougie, L. (2012). Scheduling Real-time Mixed-criticality Jobs. *IEEE Transactions on Computers*, 61(8):1140–1152.

[Becker et al., 2006] Becker, S., Hasselbring, W., Paul, A., Boskovic, M., Koziolek, H., Ploski, J., Dhama, A., Lipskoch, H., Rohr, M., Winteler, D., Giesecke, S., Meyer, R., Swaminathan, M., Happe, J., Muhle, M., and Warns, T. (2006). Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology. *SIGSOFT Software Engineering Notes*, 31(6):1 – 18.

[Benoit et al., 2006] Benoit, A., Plateau, B., and Stewart, W. J. (2006). Memory-efficient Kronecker Algorithms with Applications to the Modelling of Parallel Systems. *Future Gener. Comput. Syst.*, 22(7):838–847.

[Bèrard et al., 2001] Bèrard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., and Schnoebelen, P. (2001). *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer.

[Bernstein, 1966] Bernstein, A. (1966). Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, 15(5):757 – 763.

[Boehm et al., 1976] Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative Evaluation of Software Quality. In *Proceedings of the Second International Conference on Software Engineering*, ICSE1976, pages 592 – 605, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Boudali et al., 2008a] Boudali, H., Crouzen, P., Haverkort, B. R., Kuntz, M., and Stoelinga, M. (2008a). Arcade - A Formal, Extensible, Model-Based Dependability Evaluation Framework. In *ICECCS*, pages 243–248. IEEE Computer Society.

[Boudali et al., 2008b] Boudali, H., Crouzen, P., Haverkort, B. R., Kuntz, M., and Stoelinga, M. (2008b). Architectural Dependability Evaluation with Arcade. In *DSN*, pages 512–521.

[Boudali et al., 2007a] Boudali, H., Crouzen, P., and Stoelinga, M. (2007a). A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains. In *Proceedings of the 5th international conference on Automated technology for verification and analysis*, ATVA'07, pages 441–456, Berlin, Heidelberg. Springer-Verlag.

[Boudali et al., 2007b] Boudali, H., Crouzen, P., and Stoelinga, M. (2007b). Dynamic Fault Tree analysis using Input/Output Interactive Markov Chains. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007*, pages 708–717, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Boudali et al., 2010] Boudali, H., Crouzen, P., and Stoelinga, M. (2010). A Rigorous, Compositional, and Extensible Framework for Dynamic Fault Tree Analysis. *IEEE Trans. Dependable Sec. Comput.*, 7(2):128–143.

[Boudali et al., 2009] Boudali, H., Sözer, H., and Stoelinga, M. (2009). Architectural Availability Analysis of Software Decomposition for Local Recovery. In *SSIRI*, pages 14–22.

[Bozzano and Villafiorita, 2010] Bozzano, M. and Villafiorita, A. (2010). *Design and Safety Assessment of Critical Systems*. CRC Press (Taylor and Francis), an Auerbach Book, 1st edition.

[Brown et al., 1989] Brown, G. M., Gouda, M. G., and Wu, C.-L. (1989). Token Systems That Self-Stabilize. *IEEE Transactions on Computing*, 38(6):845 – 852.

[Buchholz, 1994] Buchholz, P. (1994). Exact and Ordinary Lumpability in Finite Markov Chains. *Journal of Applied Probability*, 31(1):59–75.

[Buchholz, 1997] Buchholz, P. (1997). Hierarchical structuring of superposed gspns. In *Petri Nets and Performance Models, 1997., Proceedings of the Seventh International Workshop on*, pages 81–90.

[Burrell et al., 2004] Burrell, J., Brooke, T., and Beckwith, R. (2004). Vineyard Computing: Sensor Networks in Agricultural Production. *IEEE Pervasive Computing*, 3:38 – 45.

[Callaway, 2009] Callaway, D. S. (2009). Tapping the Energy Storage Potential in Electric Loads to Deliver Load Following and Regulation, with Application to Wind Energy. *Energy Conversion and Management*, 50:1389 – 1400.

[Chen, 1976] Chen, P. P.-S. (1976). The Entity-relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9 – 36.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263.

[Coulouris et al., 2001] Coulouris, G., Dollimore, J., and Kindber, T. (2001). *Distributed Systems Concepts and Design*. International Computer Science Series. Addison-Wesley Pub. Co., 3 edition.

[Deb, 2001] Deb, K. (2001). *Multi-objective Optimization Using Evolutionary Algorithms*. Wiley-Interscience series in systems and optimization. John Wiley & Sons.

[Delporte-Gallet et al., 2007] Delporte-Gallet, C., Devismes, S., and Fauconnier, H. (2007). Robust Stabilizing Leader Election. In *Proceedings of the Ninth International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS2007)*, pages 219 – 233, Berlin, Heidelberg. Springer.

[Denning, 1976] Denning, P. J. (1976). Fault Tolerant Operating Systems. *ACM Computing Surveys*, 8(4):359 – 389.

[Department of Defense, 1988] Department of Defense, W. D. (1988). *Electronic Reliability Design Handbook*. Number 1. Defense Technical Information Center.

[Devismes et al., 2008] Devismes, S., Tixeuil, S., and Yamashita, M. (2008). Weak vs. Self vs. Probabilistic Stabilization. In *Proceedings of the 28th International Conference on Distributed Computing Systems (ICDCS2008)*, pages 681 – 688, Washington, DC, USA. IEEE Computer Society Press.

[Dijkstra, 1974] Dijkstra, E. W. (1974). Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643 – 644.

[D'Innocenzo et al., 2012] D'Innocenzo, A., Abate, A., and Katoen, J.-P. (2012). Robust PCTL Model Checking. In *Proceedings of the 15th ACM international conference on Hybrid Systems: Computation and Control*, HSCC '12, pages 275–286, New York, NY, USA. ACM.

[Dolev, 2000] Dolev, S. (2000). *Self-Stabilization*. The MIT Press, Cambridge, MA, USA.

[Dolev et al., 1996] Dolev, S., Gouda, M. G., and Schneider, M. (1996). Memory Requirements for Silent Stabilization. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC1996)*, pages 27 – 34, New York, NY, USA. ACM.

[Ebnenasir, 2005] Ebnenasir, A. (2005). *Automatic Synthesis of Fault-tolerance*. PhD thesis, East Lansing, MI, USA. Advisors: Sandeep S. Kulkarni, Laura Dillon, Betty Cheng, Jonathan Hall.

[Echtle, 1990] Echtle, K. (1990). *Fehlertoleranzverfahren*. Studienreihe Informatik. Springer. ISBN: 978-3-540-52680-3.

[Erlang, 1909] Erlang, A. K. (1909). The Theory of Probabilities and Telephone Conversations. *Nyt Tidsskrift for Matematik*, 20(B):33–39. accessble online via http://de.scribd.com/doc/27138581/The-Life-and-Works-of-a-K-Erlang.

[Erlang, 1917] Erlang, A. K. (1917). Solution of some Problems in the Theory of Probabilities of Significance in Automatic Telephone Exchanges. *Elektrotkeknikeren*, 13. accessble online via http://de.scribd.com/doc/27138581/The-Life-and-Works-of-a-K-Erlang.

[Feller, 1968] Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley.

[Fischer and Jiang, 2006] Fischer, M. and Jiang, H. (2006). Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *Proceedings of the Tenth International Conference on Principles of Distributed Systems*, number 4305, pages 395 – 409. Springer.

[Fränzle et al., 2011] Fränzle, M., Hahn, E. M., Hermanns, H., Wolovick, N., and Zhang, L. (2011). Measurability and Safety Verification for Stochastic Hybrid Systems. In *Proceedings of the 14th International Conference on Hybrid systems: Computation and Control (HSCC2011)*, pages 43 – 52. ACM.

[Fränzle et al., 2007] Fränzle, M., Herde, C., Teige, T., Ratschan, S., and Schubert, T. (2007). Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 1(3-4):209 – 236.

[Frolund and Koistinen, 1998a] Frolund, S. and Koistinen, J. (1998a). QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Hewlett-Packard Software Technology Laboratory.

[Frolund and Koistinen, 1998b] Frolund, S. and Koistinen, J. (1998b). Quality of Service Specification in Distributed Object Systems Design. Technical report.

[Frolund and Koistinen, 1999] Frolund, S. and Koistinen, J. (1999). Quality of Service Aware Distributed Object Systems. In *Proceedings of the Fifth USENIX Conference On Object-Oriented Technology and Systems (COOTS1999)*, pages 69 – 83.

[Fu et al., 2002] Fu, X., Bultan, T., and Su, J. (2002). Formal Verification of E-Services and Workflows. In *Proc. ESSW*, pages 188–202. Springer-Verlag.

[Garavel and Hermanns, 2002] Garavel, H. and Hermanns, H. (2002). On Combining Functional Verification and Performance Evaluation Using CADP. In *FME 2002: Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, pages 410–429.

[Garavel et al., 2001] Garavel, H., Lang, F., and Mateescu, R. (2001). An overview of CADP 2001. Research Report RT-0254, INRIA.

[Garavel et al., 2011] Garavel, H., Lang, F., Mateescu, R., and Serwe, W. (2011). CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2011*, Saabrücken, Allemagne.

[Girard and Pappas, 2005] Girard, A. and Pappas, G. J. (2005). Approximate Bisimulations for Nonlinear Dynamical Systems. In *50th IEEE Conference on Decision and Control and European Control*, pages 684 – 689, Seville, Spain. IEEE Computer Society Press.

[Golay, 1949] Golay, M. J. E. (1949). Notes On Digital Coding. *IRE*, 37.

[Graf et al., 1996] Graf, S., Steffen, B., and Lüttgen, G. (1996). Compositional Minimisation of Finite State Systems Using Interface Specifications. *Formal Asp. of Comp*, 8:607–616.

[Hamming, 1950] Hamming, R. W. (1950). Error Detecting and Error Correcting Codes. In *Bell System Technology Journal*, volume 29, pages 147 – 150.

[Hansson and Jonsson, 1994] Hansson, H. and Jonsson, B. (1994). A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:102–111.

[Hennessy and Patterson, 1996] Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann Publishers Inc.

[Hermanns, 2002] Hermanns, H. (2002). *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *Lecture Notes in Computer Science*. Springer.

[Hermanns and Katoen, 1999] Hermanns, H. and Katoen, J.-P. (1999). Automated Compositional Markov Chain Generation for a Plain-Old Telephone System. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 97–127.

[Hermanns and Katoen, 2009] Hermanns, H. and Katoen, J.-P. (2009). The How and Why of Interactive Markov Chains. In *FMCO*, pages 311–337.

[Hermanns et al., 2008] Hermanns, H., Wachter, B., and Zhang, L. (2008). Probabilistic CEGAR. In Gupta, A. and Malik, S., editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 162–175. Springer Berlin Heidelberg.

[Hillston, 1995] Hillston, J. (1995). Compositional Markovian Modelling Using a Process Algebra. In *Numerical Solution of Markov Chains*, pages 177 – 196. Kluwer Academic Publishers.

[Jalote, 1994] Jalote, P. (1994). *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Jou and Smolka, 1990] Jou, C.-C. and Smolka, S. A. (1990). Equivalences, Congruences, and Complete Axiomatizations for Probabilistic Processes. In Baeten, J. and Klop, J., editors, *CONCUR 1990 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 367–383. Springer Berlin Heidelberg.

[Juang et al., 2002] Juang, P., Oki, H., Wang, Y., Martonosi, M., Peh, L.-S., and Rubenstein, D. (2002). Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS2002)*, pages 96 – 107, New York, NY, USA. ACM.

[Kamgarpour et al., 2013] Kamgarpour, M., Ellen, C., Soudjani, S. E. Z., Gerwinn, S., Mathieux, J. L., Müllner, N., Abate, A., Callaway, D. S., Fränzle, M., and Lygeros, J. (2013). Modeling Options for Demand Side Participation of Thermostatically Controlled Loads. In *Proceedings of the IREP Symposium-Bulk Power System Dynamics and Control -IX (IREP), August 25-30, 2013, Rethymnon, Greece*.

[Katoen et al., 2007] Katoen, J.-P., Kemna, T., Zapreev, I. S., and Jansen, D. N. (2007). Bisimulation Minimisation Mostly Speeds up Probabilistic Model Checking. In *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'07, pages 87–101, Berlin, Heidelberg. Springer-Verlag.

[Katoen et al., 2005] Katoen, J.-P., Khattri, M., and Zapreev, I. S. (2005). A Markov Reward Model Checker. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, QEST '05, pages 243–, Washington, DC, USA. IEEE Computer Society.

[Katoen et al., 2012] Katoen, J.-P., Klink, D., Leucker, M., and Wolf, V. (2012). Three-Valued Abstraction for Probabilistic Systems. *Journal on Logic and Algebraic Programming*, pages 1 – 55.

[Keller, 1987] Keller, R. M. (1987). Defining operationality for explanation-based learning. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 2*, AAAI'87, pages 482–487. AAAI Press.

[Kemeny and Snell, 1976] Kemeny, J. G. and Snell, J. L. (1976). *Finite Markov Chains*. University Series in Undergraduate Mathematics. New York, NY, USA, 2, 1976 edition.

[Kharif and Pelinovsky, 2003] Kharif, C. and Pelinovsky, E. (2003). Physical Mechanisms of the Rogue Wave Phenomenon. *European Journal of Mechanics - B/Fluids*, 22(6):603 – 634.

[Koch et al., 2011] Koch, S., Mathieu, J. L., and Callaway, D. S. (2011). Modeling and Control of Aggregated Heterogeneous Thermostatically Controlled Loads for Ancillary Services. In *Proceedings of the 17th Power Systems Computation Conference*, Stockholm, Sweden.

[Kulkarni, 1999] Kulkarni, S. S. (1999). *Component Based Design of Fault-Tolerance*. PhD thesis. Advisors: Anish Arora, Mukesh Singhal, Ten-Hwang Lai.

[Kwiatkowska et al., 2002] Kwiatkowska, M., Norman, G., and Parker, D. (2002). Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *International Journal on Software Tools for Technology Transfer (STTT*, pages 52–66. Springer.

[Lamport, 1977] Lamport, L. (1977). Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125 – 144.

[Lamport, 1986a] Lamport, L. (1986a). On Interprocess Communication. Part I: Basic Formalism. *Distributed Computing*, 1(2):77 – 85.

[Lamport, 1986b] Lamport, L. (1986b). On Interprocess Communication. Part II: Algorithms. *Distributed Computing*, 1(2):86 – 101.

[Lamport, 1986c] Lamport, L. (1986c). The Mutual Exclusion Problem: Part I - A Theory of Interprocess Communication. *Journal of the ACM*, 33(2):313 – 326.

[Lamport, 2002] Lamport, L. (2002). *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Pub. Co.

[Larsen and Skou, 1989] Larsen, K. G. and Skou, A. (1989). Bisimulation Through Probabilistic Testing. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages (POPL1989*, pages 344 – 352.

[Leveson, 1995] Leveson, N. G. (1995). *Safeware : System Safety and Computers*. Addison-Wesley Pub. Co.

[Liu and Trenkler, 2008] Liu, S. and Trenkler, G. (2008). Hadamard, Khatri-Rao, Kronecker and Other Matrix Products. *International Journal of Information & Systems Sciences*, 4(1):160 – 177.

[Lowrance, 1976] Lowrance, W. W. (1976). *Of Acceptable Risk: Science and the Determination of Safety*. William Kaufmann, Inc., One First Street, Los Altos, California 94022.

[Lynch, 1996] Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Mainwaring et al., 2002] Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., and Anderson, J. (2002). Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA2002)*, pages 88 – 97, New York, NY, USA. ACM.

[Malhamè and Chong, 1985] Malhamè, R. and Chong, C.-Y. (1985). Electric-load Model Synthesis by Diffusion Approximation of a High-order Hybrid-state Stochastic-system. *IEEE Transactions on Automatic Control*, 30:854 – 860.

[Manna and Pnueli, 1981a] Manna, Z. and Pnueli, A. (1981a). *Temporal Verification of Concurrent Programs: The Temporal Framework for Concurrent Programs*, lecture notes in computer science 5, pages 215 – 271. The Correctness Problem in Computer Science. Academic Press.

[Manna and Pnueli, 1981b] Manna, Z. and Pnueli, A. (1981b). Verification of Concurrent Programs Part I: The Temporal Framework. Technical Report STAN-CS-82-915, Stanford University, Stanford, CA, USA.

[Mertsiotakis, 1998] Mertsiotakis, V. (1998). *Approximate Analysis Methods for Stochastic Process Algebras*. PhD thesis, Universität Erlangen-Nürnberg. Advisors: Gerhard Herold, Ulrich Herzog, Manuel Silva.

[Meyer, 2009] Meyer, R. (2009). *Structural Stationarity of the π-Calculus*. PhD thesis, Department für Informatik, Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften, Carl von Ossietzky Universität Oldenburg.

[Milner, 1999] Milner, R. (1999). *Communicating and Mobile Systems - the π-calculus*. Cambridge University Press.

[Milner et al., 1992] Milner, R., Parrow, J., and Walker, D. (1992). A Calculus of Mobile Processes, I. *Information adn Comutation*, 100:1 – 40.

[Mitrohin and Podelski, 2011] Mitrohin, C. and Podelski, A. (2011). Composing Stability Proofs for Hybrid Systems. In *FORMATS*, pages 286–300.

[Moore, 1965] Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*, 38(8).

[Müllner, 2007] Müllner, N. (2007). Simulation of Self-Stabilizing Distributed Algorithms to Determine Fault Tolerance Measures. Diplomarbeit, Department für Informatik, Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften, Carl von Ossietzky Universität Oldenburg, Oldenburg (Oldb), Germany.

[Müllner et al., 2008] Müllner, N., Dhama, A., and Theel, O. (2008). Derivation of Fault Tolerance Measures of Self-Stabilizing Algorithms by Simulation. In *Proceedings of the 41st Annual Symposium on Simulation (AnSS2008)*, pages 183 – 192, Ottawa, ON, Canada. IEEE Computer Society Press.

[Müllner et al., 2009] Müllner, N., Dhama, A., and Theel, O. (2009). Deriving a Good Trade-off Between System Availability and Time Redundancy. In *Proceedings of the Symposia and Workshops on Ubiquitous, Automatic and Trusted Computing*, number E3737 in Track "International Symposium on UbiCom Frontiers - Innovative Research, Systems and Technologies (Ufirst-09)", pages 61 – 67, Brisbane, QLD, Australia. IEEE Computer Society Press.

[Müllner and Theel, 2011] Müllner, N. and Theel, O. (2011). The Degree of Masking Fault Tolerance vs. Temporal Redundancy. In *Proceedings of the 25th IEEE Workshops of the International Conference on Advanced Information Networking and Applications (WAINA2011)*, Track "The Seventh International Symposium on Frontiers of Information Systems and Network Applications (FINA2011)", pages 21 – 28, Biopolis, Singapore. IEEE Computer Society Press.

[Müllner et al., 2012] Müllner, N., Theel, O., and Fränzle, M. (2012). Combining Decomposition and Reduction for State Space Analysis of a Self-Stabilizing System. In *Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications (AINA2012)*, pages 936 – 943, Fukuoka-shi, Fukuoka, Japan. IEEE Computer Society Press. Best Paper Award.

[Müllner et al., 2013] Müllner, N., Theel, O., and Fränzle, M. (2013). Combining Decomposition and Reduction for the State Space Analysis of Self-Stabilizing Systems. In *Journal of Computer and System Sciences (JCSS)*, volume 79, pages 1113 – 1125. Elsevier Science Publishers B. V. The paper is an extended version of a publication with the same title.

[Müllner et al., 2014a] Müllner, N., Theel, O., and Fränzle, M. (2014a). Combining Decomposition and Lumping to Evaluate Semi-hierarchical Systems. In *Proceedings of the 28th IEEE International Conference on Advanced Information Networking and Applications (AINA2014)*.

[Müllner et al., 2014b] Müllner, N., Theel, O., and Fränzle, M. (2014b). Composing Thermostatically Controlled Loads to Determine the Reliability against Blackouts. In *Proceedings of the 10th International Symposium on Frontiers of Information Systems and Network Applications (FINA2014)*.

[Musa et al., 1987] Musa, J. D., Iannino, A., and Okumoto, K. (1987). *Software Reliability – Measurement, Prediction, Application*. McGraw-Hill, New York, NY, USA.

[Nesterenko and Tixeuil, 2011] Nesterenko, M. and Tixeuil, S. (2011). Ideal Stabilization. In *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA2011)*, pages 224 – 231, Biopolis, Singapore. IEEE Press. Best Paper Award.

[Neumann, 2000] Neumann, P. G. (2000). Practical Architectures for Survivable Systems and Networks. Report 2, SRI International, SRI International, Room EL243, 333 Ravenswood Avenue, Menlo Park CA 94025-3493.

[Norris, 1998] Norris, J. (1998). *Markov Chains*. Number Nr. 2008 in Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

[Owicki and Gries, 1976] Owicki, S. and Gries, D. (1976). Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19:279 – 285.

[Owicki and Lamport, 1982] Owicki, S. and Lamport, L. (1982). Proving Liveness Properties of Concurrent Programs. volume 4, pages 455 – 495, New York, NY, USA. ACM.

[Patterson and Hennessy, 2005] Patterson, D. A. and Hennessy, J. L. (2005). *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc.

[Pfeiffer, 1978] Pfeiffer, P. E. (1978). *Concepts of Probability Theory*. Dover Books on Mathematics. Dover Publications.

[Pfleeger, 1997] Pfleeger, C. P. (1997). *Security in Computing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Pnueli and Zuck, 1986] Pnueli, A. and Zuck, L. (1986). Verification of Multiprocess Probabilistic Protocols. *Distributed Computing*, 1(1):53 – 72.

[Podelski and Wagner, 2006] Podelski, A. and Wagner, S. (2006). Model Checking of Hybrid Systems: From Reachability Towards Stability. In *HSCC*, pages 507–521.

[Pucella, 2000] Pucella, R. (2000). Review of Communicating and Mobile Systems: The π-calculus. In [Milner, 1999], pages I–XII, 1–161.

[Rakow, 2011] Rakow, A. (2011). *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, Department für Informatik, Fakultät II - Informatik, Wirtschafts- und Rechtswissenschaften, Carl von Ossietzky Universität Oldenburg, Uhlhornsweg 49-55, 26129 Oldenburg, Germany. Advisors: Eike Best, Ernst-Rüdiger Olderog.

[Rozenberg and Vaandrager, 1996] Rozenberg, G. and Vaandrager, F. W., editors (1996). *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*. Springer.

[Rus et al., 2003] Rus, I., Komi-Sirvio, S., and Costa, P. (2003). Software Dependability Properties - A Survey of Definitions, Measures and Techniques. Survey 03-110, Fraunhofer USA - Center for Experimental Software Engineering, Maryland, Fraunhofer Center - Maryland, University of Maryland, 4321 Hartwick Road, Suite 500, College Park, MD 20742.

[Sarkar, 1993] Sarkar, V. (1993). A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs. In Banerjee, U., Nicolau, D. G. A., and Padua, D., editors, *Languages and Compilers for Parallel Computing (LCPC)*, volume 757 of *Lecture Notes in Computer Science*, pages 16–30. Springer Berlin Heidelberg.

[Schneider, 1998] Schneider, F. B. (1998). *Trust in Cyberspace*. National Academy Press, Washington, DC, USA.

[Schneider, 1993] Schneider, M. (1993). Self-stabilization. *ACM Computing Surveys*, 25(1):45 – 67.

[Schroeder and Gibson, 2007] Schroeder, B. and Gibson, G. A. (2007). Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the Fifth USENIX conference on File and Storage Technologies (FAST2007)*, page 1, Berkeley, CA, USA. USENIX Association.

[Schroeder et al., 2009] Schroeder, B., Pinheiro, E., and Weber, W.-D. (2009). DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS2009)*, pages 193 – 204, New York, NY, USA. ACM.

[Shanks, 1985] Shanks, D. (1985). *Solved and Unsolved Problems in Number Theory*. Chelsea Publishing Co., Inc., New York, USA.

[Shemer and Sergeeva, 2009] Shemer, L. and Sergeeva, A. (2009). An Experimental Study of Spatial Evolution of Statistical Parameters in a Unidirectional Narrow-banded Random Wavefield. *Journal of Geophysical Research*, 114.

[Sistla, 1985] Sistla, A. P. (1985). On Characterization of Safety and Liveness Properties in Temporal Logic. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC1985)*, pages 39 – 48, New York, NY, USA. ACM.

[Smith, 2003] Smith, G. (2003). Probabilistic Noninterference through Weak Probabilistic Bisimulation. In *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, pages 3 – 13.

[Sommerville, 2004] Sommerville, I. (2004). *Software Engineering*. Pearson Addison Wesley, seventh edition.

[Sonneborn and van Vleck, 1964] Sonneborn, L. M. and van Vleck, F. S. (1964). The Bang-bang Principle for Linear Control Systems. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 2:151–159.

[Soudjani and Abate, 2013a] Soudjani, S. E. Z. and Abate, A. (2013a). Adaptive and Sequential Gridding Procedures for the Abstraction and the Verification of Stochastic Processes. Submitted for review to the Society for Industrial and Applied Mathematics (SIAM).

[Soudjani and Abate, 2013b] Soudjani, S. E. Z. and Abate, A. (2013b). Aggregation of Thermostatically Controlled Loads by Formal Abstractions. *Prodceedings of the Eurpoean Control Conference (ECC2013)*. submitted for review.

[Soudjani and Abate, 2013c] Soudjani, S. E. Z. and Abate, A. (2013c). Probabilistic Reachability Computation for Mixed Deterministic-Stochastic Processes. unpublished draft.

[Stark and Einaudi, 1996] Stark, D. and Einaudi, M. (1996). *Heterarchy: Asset Ambiguity, Organizational Innovation, and the Postsocialist Firm*. Working Papers on Transitions From State Socialism. Center for Advanced Human Resource Studies, Cornell University, ILR School.

[Storey, 1996] Storey, N. R. (1996). *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Tanenbaum and Steen, 2001] Tanenbaum, A. S. and Steen, M. V. (2001). *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

[Theel, 2000] Theel, O. (2000). Exploitation of Ljapunov Theory for Verifying Self-Stabilizing Algorithms. In Herlihy, M., editor, *Distributed Computing*, volume 1914 of *Lecture Notes in Computer Science*, pages 213 – 251. Springer.

[Tixeuil, 2009] Tixeuil, S. (2009). *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1 – 26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group. http://www.crcpress.com/product/isbn/9781584888185.

[Trivedi, 2002] Trivedi, K. S. (2002). *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. John Wiley & Sons, second edition.

[Williams and Sunter, 2000] Williams, T. W. and Sunter, S. (2000). How Should Fault Coverage Be Defined? In *Proceedings of the 18th IEEE VLSI Test Symposium*, VTS '00, page 325, Washington, DC, USA. IEEE Computer Society.

[Wirth, 1995] Wirth, N. (1995). A Plea for Lean Software. *Computer*, 28(2):64 – 68.

[Zhang et al., 2010] Zhang, L., She, Z., Ratschan, S., Hermanns, H., and Hahn, E. M. (2010). Safety Verification for Probabilistic Hybrid Systems. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV2010)*, pages 196 – 211.

List of Figures

A. Appendix

A.1 Employed resources

In the writing of the present thesis, the following software has been used: LaTeX(MikTex, Texnic Center), Open Office, Microsoft Office, MatLab (including the functions fig2u3d and plot2svg), Prism, Dia, and Inkscape (adapted to include Latex commands). The wiss-doc package by Roland Bless has been slightly adapted to build the present thesis. The computing resources have been provided by the Carl von Ossietzky Universität Oldenburg and the OFFIS Institute for Computer Science. This work was partly supported (presented in chronologically descending order) by the German Research Council (DFG) as part of the Transregional Collaborative Research Center Automatic Verification and Analysis of Complex Systems (SFB/TR 14 AVACS), the Graduate College on Trustworthy Software Systems TrustSoft (GRK 1076/1), the European Commission under the MoVeS project (FP7-ICT-2009-257005) and by the funding initiative *Niedersächsisches Vorab* of the Volkswagen Foundation and the Ministry of Science and Culture of Lower Saxony as part of the *Interdisciplinary Research Center on Critical Systems Engineering for Socio-Technical Systems*.

A.2 List of abbreviations

abbreviation	meaning	first occurrence
BASS	Self Stabilizing Broadcast Algorithm	page 57
DTMC	Discrete Time Markov Chain	page 14
IWA	Instantaneous Window Availability	page 49
LTP	Law of Total Probability	page 68
LWA	Limiting Window Availability	page 47
MOOp	Multi Objective Optimization	page 137
PCTL	Probabilistic Real Time Computation Tree Logic	page 47
QoS	Quality of Service	page 160
ROM	Read Only Memory	page 8
SRAM	Static Random Access Memory	page 8
TLA	Traffic Light Algorithm	page 17

Table A.1: List of abbreviations

A.3   Table of notation

symbol	formula example	description	first occurrence
$\mathcal{P}$	$s \models \mathcal{P}$	(safety) predicate	page 28
$\sigma_{t,k}^i$	$\sigma = \langle s_0 \xrightarrow{\pi_i,p} s_1 \ldots \rangle$	partial execution trace	page 41
$\diamond$	$\models \varphi(\bar{x}) \subset \square w$	eventually	page 47
$A(t)$	$pr(s_{i,t} \models \mathcal{P})$	point availability	page 33
$A(t)$ $t \to \infty$		limiting availability	page 33
$\mathcal{S}_{legal}$		set of legal states	page 34
$\mathfrak{S}$	$\mathfrak{S} = \{\Pi, E, \mathfrak{A}, \mathfrak{s}\}$	system	page 6
Π	$\Pi = \{\pi_1, \ldots, \pi_n\}$	set of n processes	page 5
E	$E = \{e_{i,j}, \ldots\}$	communication channels	page 5
$\mathfrak{A}$	$\mathfrak{A} = \{a_1, \ldots\}$	algorithm	page 5
	$a_k : g_k \to c_k$	guarded command	page 6
a_k		label	page 6
g_k	$\langle R_1, \ldots, R_n \rangle$	guard	page 6
c_k	$c_k : R_i := value$	command	page 6
s		system state	page 6
$\mathcal{S}$		state space	page 6
$\mathfrak{s}_i$		selection probability of π_i	page 12
$\mathbf{q}$	$\mathbf{q} = \mathfrak{s}_i \cdot pr(q_i)$	combined success probability	page 13
$\mathbf{p}$	$\mathbf{p} = \mathfrak{s}_i \cdot p$	combined error probability	page 13
$pr(\overrightarrow{s_i, s_j})$		transition probability	page 14
$\mathcal{D}$		discrete time Markov chain	page 15
$\sigma_{t,k}^i$		partial execution trace	page 13
l_w		limiting window availability	page 47
w		window size	page 47
v		LWA vector	page 48
g		LWA vector gradient	page 49
$\mathcal{D}_{\text{LWA}}$		DTMC computing the LWA	page 51
$\mathcal{D}'$		maximally lumped DTMC	page 67
$\overline{\mathcal{D}'}$		partially lumped DTMC	page 90
$\mathcal{D}'_{\text{LWA}}$		lumped DTMC computing the LWA	page 70
$\sim$	$s_i \sim s_j, [s_i]_\sim$	equivalence relation	page 65
τ	$\tau(\mathfrak{S}) = \{\Pi_1, \Pi_2 \ldots\}$	set of subsystems	page 76
$\otimes$	$\mathcal{D}' = \mathcal{D}'_1 \otimes \ldots \otimes \mathcal{D}'_n$	recomposition/uncoupling operator	page 76

Table A.2: Table of symbols

A.4   Definitions

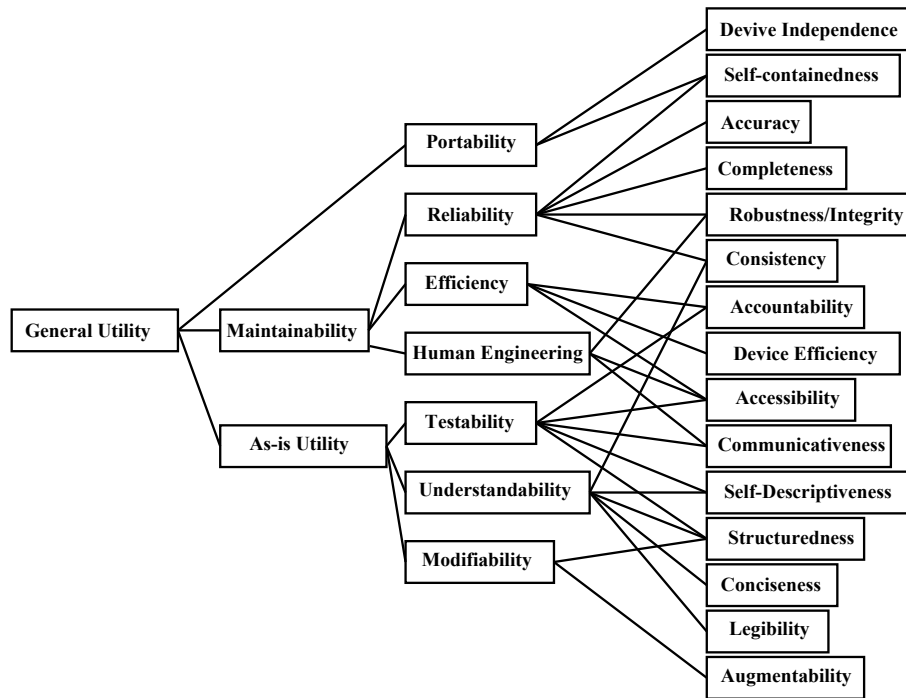## A.4.1   Fault tolerance trees

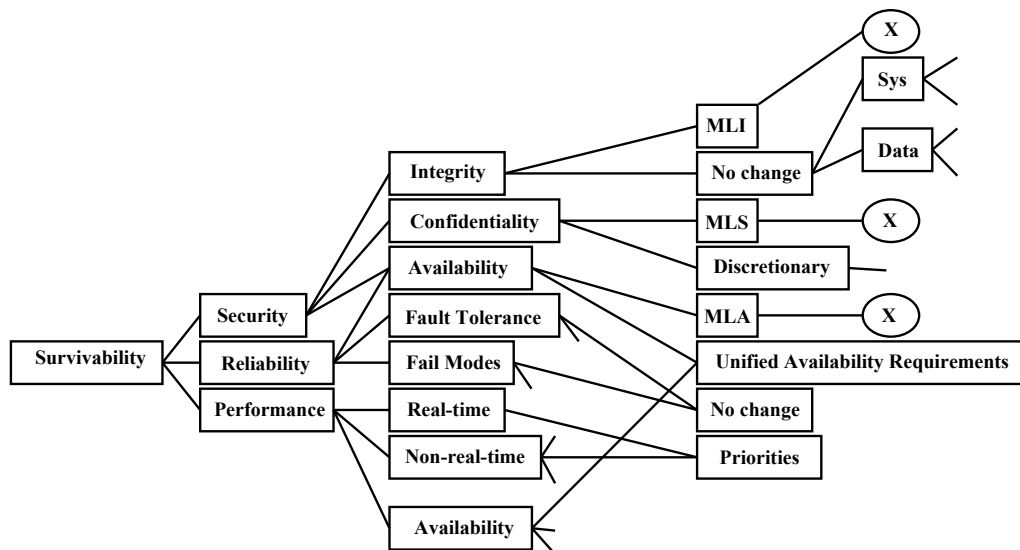Figure A.1: Software quality characteristics tree by Boehm et al. [Boehm et al., 1976, p.595]

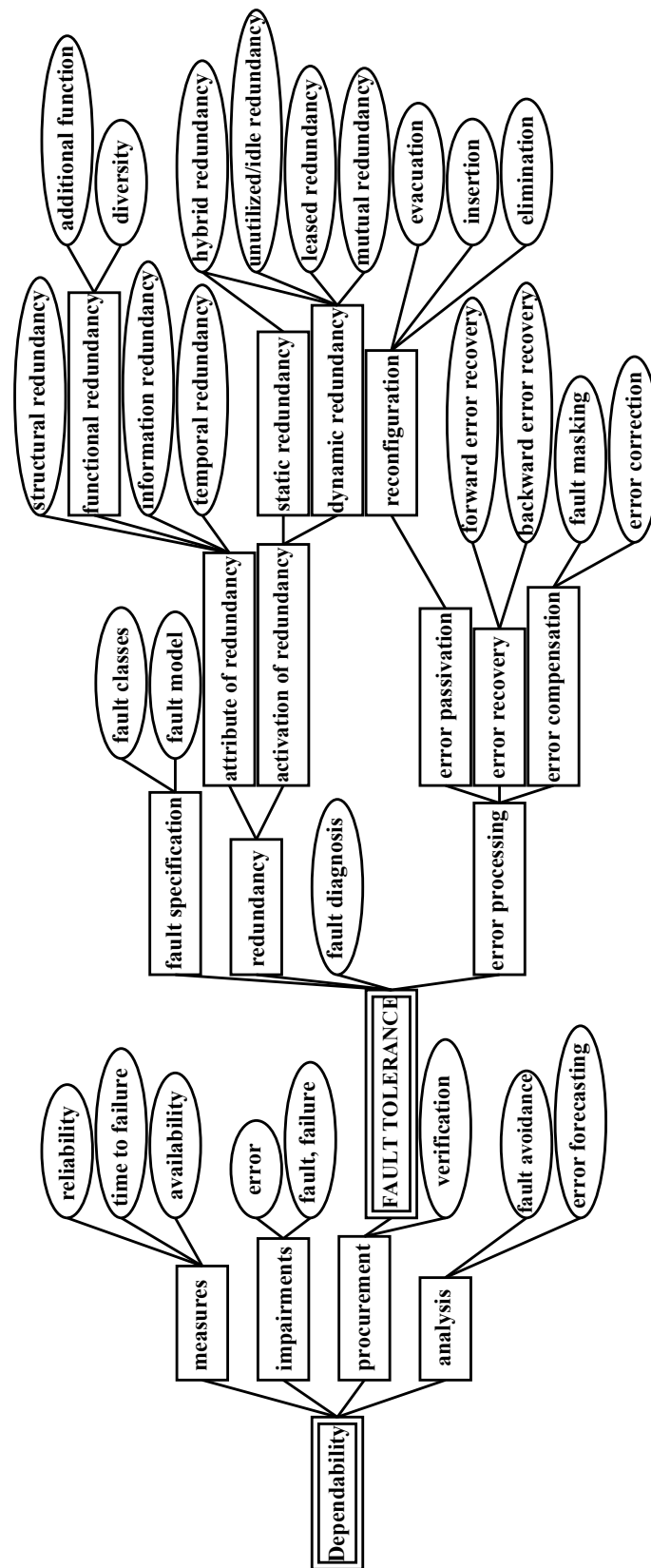Figure A.3: Illustrative subset of requirements hierarchy [Neumann, 2000, p.51]

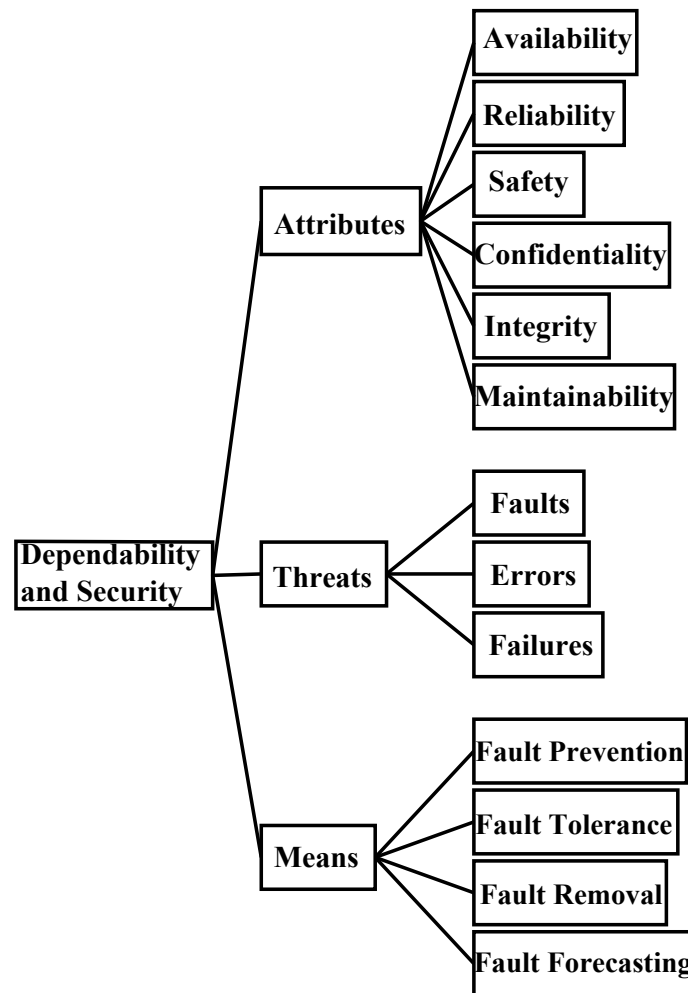Figure A.2: Dependability tree by Echtle [Echtle, 1990]

Figure A.4: Dependability tree by Avižienis et al. [Avižienis et al., 2004, p.14]

A.4.2 Fault tolerance

Fault tolerance means to avoid service failures in the presence of faults. [Avižienis et al., 2004]

A system can provide its services even in the presence of faults. [Tanenbaum and Steen, 2001]

A system is fault tolerant if it can mask the presence of faults in the system by using redundancy. The goal of fault tolerance is to avoid system failure, even if faults are present. [Jalote, 1994]

Fault tolerance is an approach by which reliability of a computer system can be increased beyond what can be achieved by traditional methods. Fault tolerant systems employ redundancy to mask various types of failures. [Jalote, 1994]

A fault tolerant service [. . .] always guarantees strictly correct behavior despite a certain number and type of faults. [Coulouris et al., 2001]

Fault tolerance (or graceful degradation) is the capacity of a system to operate properly on the hypothesis of the failure of one (or more) of its components. [Bozzano and Villafiorita, 2010, p.34]

The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. [ISO, 2001]

By quality of service we refer to non-functional properties such as

performance,	reliability,	availability,			and security.	[Frolund and Koistinen, 1998a]
performance,	reliability,		quality of data,	timing,	and security.	[Frolund and Koistinen, 1998b]
performance,	reliability,	availability,		timing,	and security.	[Frolund and Koistinen, 1999]

A.4.3 Safety

For $\mathcal{P}$ to be a safety property, if $\mathcal{P}$ does not hold for an execution then at some point some "bad thing" must happen. Such a "bad thing" must be irremediable because a safety property states that the "bad thing" never happens during execution. Thus, $\mathcal{P}$ is a safety property if and only if

$$(\forall \sigma : \sigma \in \mathcal{S}^\omega : \sigma \not\models \mathcal{P} \Rightarrow (\exists i : 0 \leq i : (\forall \beta : \beta \in \mathcal{S}^\omega : \sigma_i \beta \not\models \mathcal{P}))). \qquad (A.1)$$

[Alpern and Schneider, 1985]

Examples of safety properties include mutual exclusion, deadlock freedom, partial correctness, and first-come-first-serve. In *mutual exclusion*, the proscribed "bad thing" is two processes executing in critical sections at the same time. In deadlock freedom it is deadlock. In *partial correctness* it is terminating in a state not satisfying the postcondition after having been started in a state that satisfies the precondition. Finally, in *first-come-first-serve*, which states that requests are serviced in the order they are made, the "bad thing" is servicing a request that was made after one that was not yet being serviced. [Alpern and Schneider, 1985]

Consider first the class of program properties that hold continuously throughout the execution. They are expressible by formulas of the form:

$$| \equiv \Box w.$$

Such a formula states that $\Box w$ holds for every admissible computation, i.e. w is an invariant of every computation. By generalization rule this could have been written as $| \equiv w$, but we prefer the above form since it emphasizes that we are discussing invariance properties.

Note that the initial condition associated with the admissible computation is:

$$at\bar{l}_0 \wedge \bar{y} = f_0(\bar{x}) \wedge \varphi(\bar{x})$$

which characterizes the initial state for input $\bar{x}$ satisfying the precondition $\varphi(\bar{x})$. Here, $\bar{l}_0 = (\bar{l}_0^1, \dots, \bar{l}_0^m)$ is the set of initial locations in each of the processes. To emphasize the precondition $\varphi(\bar{x})$ we sometimes express $| \equiv \Box w$ as

$$| \equiv \varphi(\bar{x}) \subset \Box w.$$

A formula in this form therefore expresses an *invariance property*. The properties in this class are also known as *safety properties*, based on the premise that they ensure that "nothing bad will happen" [Lamport, 1977]. [Manna and Pnueli, 1981a, p.252]

A safety property is one which states that something will not happen. For example, the partial correctness of a single process program is a safety property. It states that if the

program is started with the correct input, then it cannot stop if it does not produce the correct output. [Lamport, 1977]

Formally, safety property $\mathcal{P}$ is defined as an LT property over AP such that any infinite word σ where $\mathcal{P}$ does not hold contains a *bad prefix*. The latter means a finite prefix $\widehat{\sigma}$ where the bad thing has happened, and thus no infinite word that starts with this prefix $\widehat{\sigma}$ fulfills $\mathcal{P}$. [Baier and Katoen, 2008, p.112]

A safety property expresses that, under certain conditions, an event never occurs.[Bèrard et al., 2001]

Safety is a property of a system that it will not endanger human life or the environment. [Storey, 1996]

The term safety critical system is normally used as a synonym for a safety-related system, although in some cases it may suggest a system of high criticality. [Storey, 1996]

We will define safety as a judgment of the acceptability of risk, and risk, in turn, as a measurement of the probability and the severity of harm to human health. A thing is safe if its attendant risks are judged to be acceptable. [Lowrance, 1976]

Safety of a system is the absence of catastrophic consequences on the user(s) and the environment. [Avižienis et al., 2004]

Safety-critical software is any software that can directly or indirectly contribute to the occurrence of a hazardous system state. [Leveson, 1995]

Safety-critical functions are those system functions whose correct operation, incorrect operation (including correct operation at the wrong time), or lack of operation could contribute to a system hazard. [Leveson, 1995]

Safety can be described as a characteristic of the system of not endangering, or causing harm to, human lives or the environment in which the equipment or plant operates. That is, safety evaluates the system operation in terms of freedom from occurrence of catastrophic failures. [Bozzano and Villafiorita, 2010]

A.4.4 Fairness

[. . .] By that we mean that no process which is ready to run (i.e. enabled) will be neglected forever. Stated more precisely, we exclude infinite executions in which a certain process which has not terminated is never scheduled from a certain point on. Note that all finite terminating sequences are necessarily fair. [Manna and Pnueli, 1981a, p.246]

Weak fairness of A [action] asserts that an A step must eventually occur if A is continuously enabled. [Lamport, 2002]

Strong fairness of A asserts that an A step must eventually occur if A is continually enabled. Continuously means without interruption. Continually means repeatedly, possibly with interruptions. [Lamport, 2002]

For a transition system $TS = (\mathcal{S}, Act, \rightarrow, I, AP, L)$ without terminal states, $A \subseteq Act$, and infinite execution fragment $\rho = s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots$ of TS:

1. ρ is *unconditionally A-fair* whenever $\overset{\infty}{\exists} j.\ \alpha_j \in A$.

2. ρ is *strongly* A-fair whenever

$$(\overset{\infty}{\exists} j.Act(s_j) \cap A \neq \varnothing) \Rightarrow (\overset{\infty}{\exists} j.\alpha_j \in A).$$

3. ρ is *weakly* A-fair whenever

$$(\overset{\infty}{\forall} j.Act(s_j) \cap A \neq \varnothing) \Rightarrow (\overset{\infty}{\exists} j.\alpha_j \in A).$$

[Baier and Katoen, 2008, p.130]

Let TS be a transition system with the set of actions Act and $\mathcal{F}$ a fairness assumption for Act. $\mathcal{F}$ is called *realizable* for TS if for every reachable state $s : FairPaths_{\mathcal{F}}(s) \neq \varnothing$. [Baier and Katoen, 2008, p.139]

A.4.5   Liveness

A partial execution α is *live* for a property $\mathcal{P}$ if and only if there is a sequence of states β such that $\alpha\beta \models \mathcal{P}$. A *liveness property* is one for which every partial execution is live. Thus, $\mathcal{P}$ is a liveness property if and only if

$$(\forall \alpha : \alpha \in \mathcal{S}^* : (\exists \beta : \beta \in \mathcal{S}^{\Omega} : \alpha\beta \models \mathcal{P})) \tag{A.2}$$

[Alpern and Schneider, 1985]

Examples of liveness properties include starvation freedom, termination, and guaranteed service. In *starvation freedom* (i.e. the dining philosophers problem), which states that a process makes progress infinitely often, the "good thing" is making progress. In *termination*, which asserts that a program does not run forever, the "good thing" is completion of the final instruction. Finally, in a *guaranteed service*[1], which states that every request for service is satisfies eventually, the "good thing" is receiving service. [Alpern and Schneider, 1985]

$$(\exists \beta : \beta \in \mathcal{S}^{\Omega} : (\forall \alpha : \alpha \in \mathcal{S}^* : \alpha\beta \models \mathcal{P})). \tag{A.3}$$

$\mathcal{P}$ is a uniform liveness property if and only if there is a single execution (β) that can be appended to every partial execution (α) so that the resulting sequence is in $\mathcal{P}$. [Alpern and Schneider, 1985]

$$\begin{aligned}(\exists \gamma : \gamma \in \mathcal{S}^{\Omega} : \gamma \models \mathcal{P}) \wedge \\ (\forall \beta : \beta \in \mathcal{S}^{\Omega} : \beta \models \mathcal{P} \Rightarrow (\forall \alpha : \alpha \in \mathcal{S}^* : \alpha\beta \models \mathcal{P}))\end{aligned} \tag{A.4}$$

$\mathcal{P}$ is an *absolute-liveness* property if and only if it is non-empty and any execution (β) in $\mathcal{P}$ can be appended to any partial execution (α) to obtain a sequence in $\mathcal{P}$. [Sistla, 1985][2]

A second[3] category of properties are those expressible by formulas of the form

$$| \equiv w_1 \supset \Diamond w_2$$

---

[1] This is called responsiveness in [Manna and Pnueli, 1981b]

[2] This definition is also published by Alpern & Schneider [Alpern and Schneider, 1985]

[3] The first category are invariance, i.e. safety properties as described in [Manna and Pnueli, 1981a, p.252].

This formula states that for every admissible computation, if w_1 is initially true then w_2 must eventually be realized. In comparison with invariance properties that only describe the preservation of a desired property from one step to the next, an eventuality property guarantees that some event will finally be accomplished. It is therefore more appropriate for the statement of goals which need many steps for their attainment.

Note that because of the suffix closure of the set of admissible computations this formula is equivalent to:
$$| \equiv \Box(w_1 \supset \Diamond w_2)$$
which states that whenever w_1 arises during the computation it will eventually be followed be the realization of w_2.

A property expressible by such formula is called an *eventuality* (*liveness*) *property* [Owicki and Lamport, 1982, Owicki and Gries, 1976]. [Manna and Pnueli, 1981a, p.260][4]

A liveness property is one which states that something must happen. An example of a liveness property is the statement that a program will terminate if its input is correct. [Lamport, 1977]

LT property $\mathcal{P}_{live}$ over AP is a *liveness* property whenever $pref(\mathcal{P}_{live}) = (2^{AP})^*$. [Baier and Katoen, 2008, p.121]

A liveness property states that, under certain conditions, some event will ultimately occur. [Bèrard et al., 2001]

A.4.6 Threats to system safety

An incorrect step, process, or data in a computer program. [IEE, 1990]

A fault is the (adjudged or hypothesized) cause of an error. When it produces an error, it is active, otherwise it is dormant. [Avižienis et al., 2001]

An error is that part of the system state that may cause subsequent failure. Before an error is detected (by the system), it is latent. The detection of an error is indicated at the service interface by an error message or error signal. [Avižienis et al., 2001]

A failure of a system is an event that corresponds to a transition from correct service to incorrect service. It occurs when an error reaches its service interface. The inverse transition is called service restoration. [Avižienis et al., 2004]

$$\ldots fault \xrightarrow{activation} error \xrightarrow{propagation} failure \xrightarrow{causation} fault \ldots$$

[Avižienis et al., 2001, p.3]

---
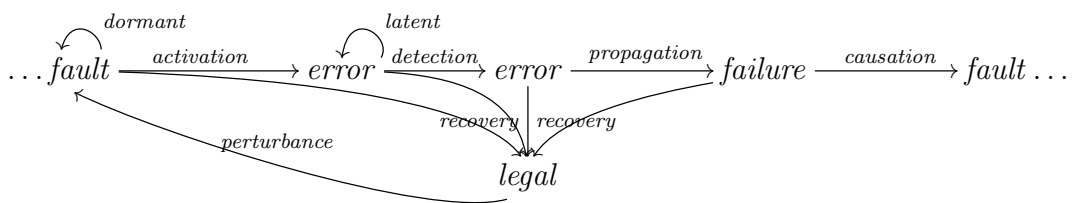
[4]Manna and Pnueli refer to a preliminary draft of [Owicki and Lamport, 1982].

A.4.7 Availability

Quality attributes are detailed quality properties of a software system, that can be measured using a quality metric. A detailed metric is a measurement scale combined with a fixed procedure describing how measurement is to be conducted. The application of a quality metric to a specific software-intensive system yields a measurement. [ISO, 1999]

Probability, that a system will work without failures at any time point t. [Echtle, 1990]

Availability is the property asserting that a resource is usable or operational during a given time period, despite attacks or failures. [Schneider, 1998]

Now we define the instantaneous availability (or point availability) $A(t)$ of a component (or a system) as the probability that the component is properly functioning at time t, that is, $A(t) = P(I(t) = 1)$. [Trivedi, 2002]

[. . .] we define the limiting or steady state availability (or simply availability) of a component (system) as the limiting value of $A(t)$ as t approaches infinity. [Trivedi, 2002]

The interval (or average) availability is the expected fraction of time a component (system) is up in a given interval. [Trivedi, 2002]

Availability is the expected fraction of time during which a software component or system is functioning acceptably. [Musa et al., 1987]

The availability of a system is the probability that the system will be functioning correctly at a given time. [. . .] Availability presents a fraction of time for which a system is functioning correctly. [Storey, 1996]

Availability means that assets are available to authorized parties at appropriate times. In other words, if some person or system has legitimate access to a particular set of objectives, that access should not be prevented. For this reason, availability is sometimes known by its opposite, denial of service. [Pfleeger, 1997]

Availability is the probability that a system, at a point in time, will be operational and able to deliver the requested services. [Sommerville, 2004]

Availability is a system's readiness for correct service. [. . .] Availability presents a fraction of time for which a system is functioning correctly, where correct service is delivered when the service implements the system function. [Avižienis et al., 2004]

Availability evaluates the probability of a system to operate correctly at a specific point in time. [. . .] Alternatively, availability can be seen as measuring the percentage of time the system is providing correct service over a given time interval. [Bozzano and Villafiorita, 2010]

A.4.8 Reliability

The probability that the component survives until some time t is called reliability $R(t)$ of the component. [Trivedi, 2002, p.124]

The probability of a failure-free operation over a specified time in a given environment for a specific purpose. [Sommerville, 2004]

Reliability refers to the characteristic of a given system of being able to operate correctly over a given period of time. That is, reliability evaluates the probability that the system

will function correctly when operating for a time interval t. [...] Equivalently, reliability can be defined in terms of *failure rate*, that is, the rate at which system components fail; or *time to failure*, that is, the time interval between beginning of system operation and occurrence of the first fault. [Bozzano and Villafiorita, 2010]

Mission reliability is the measure of the ability of an item to perform its required function for the duration of a specified mission profile. It defines that the system will not fail to complete the mission, considering all possible redundant modes of operation. [Department of Defense, 1988]

It is the probability of failure-free operation of a computer program for a specified time in a specified environment. [Musa et al., 1987]

The probability that software will not cause the failure of a system for a specified time, under specified conditions. [IEE, 1988]

The ability of a system or component to perform its required functions under stated conditions for a specified period of time. [IEE, 1990]

The capability of the software product to maintain a specified level of performance when used under specific conditions. [ISO, 2001]
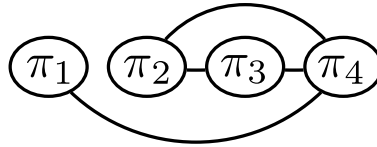
A.5 Source code
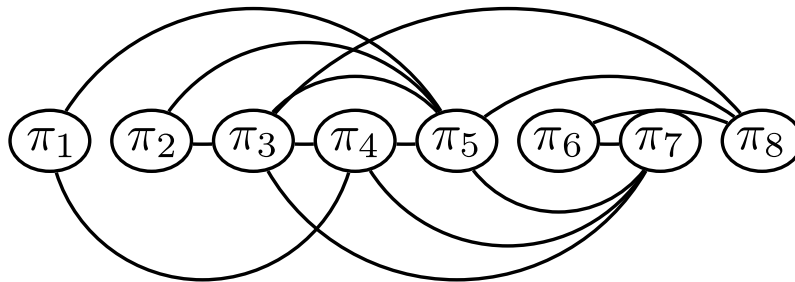
A.5.1 Simulation

Figure A.5: Four process system

Figure A.6: Eight process system

A.5.2 The BASS example

The required source code is available at http://www.mue-tech.com/BASS.zip. The required tools are MatLab and a tool to view open document tables (e.g. Libre, Open or Microsoft Office). The file bass1.ods contains all matrices. The variables in the symbolic $\mathcal{M}_1$ are replaced by their numerical values. MatLab accomplishes this as shown in file M1.m (cf. line 26). The stationary distribution is computed with the code in lines 27 to 29. The formulas in the cells of the table show how $\mathcal{M}_1$ is uncoupled to $\mathcal{M}_{1,-}$ and $\mathcal{M}_{\pi_4}$. It further shows the lumping of $\mathcal{M}_{1,-}$ to $\mathcal{M}'_{1,-}$. The next step is the symbolic construction of $\mathcal{M}_2$ and the substitution of the variables with numerical values. Analogously to the root sub-Markov chain, file M2.m provides the i) symbolic sub-Markov chain, the variable substitution and the computation of the stationary distribution. The lumping of $\mathcal{M}_2$ is shown in the table again. Finally, the file Recomposition.m composes the two sub-Markov chains, computes the stationary[5] distribution. For comparison, the script for the full product chain is included in the file RecompositionFULL.m.

Figure A.7 shows the color plots of the corresponding Markov chains. For the color table please refer to figure 7.9 on page 121.

---

[5]Previously, it was argued that the stationary distribution can as well be computed from the stationary distributions of the particular sub-Markov chains. Yet, with a numerical computation rather than a symbolic solving, computing the stationary distribution is only a matter of seconds (as discussed in section 6.5.1).
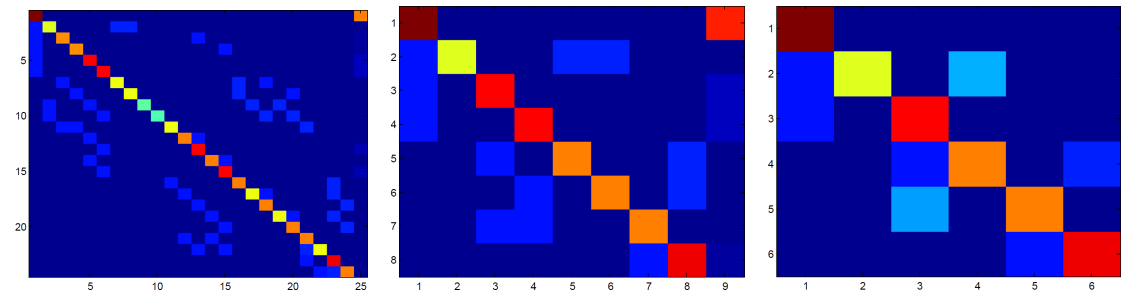
(a) Root transition matrix $\mathcal{D}_1$ from the BASS Example

(b) Uncoupled root transition matrix $\mathcal{D}_{1,-}$ from the BASS Example

(c) Lumped and uncoupled root transition matrix $\mathcal{D}'_{1,-}$
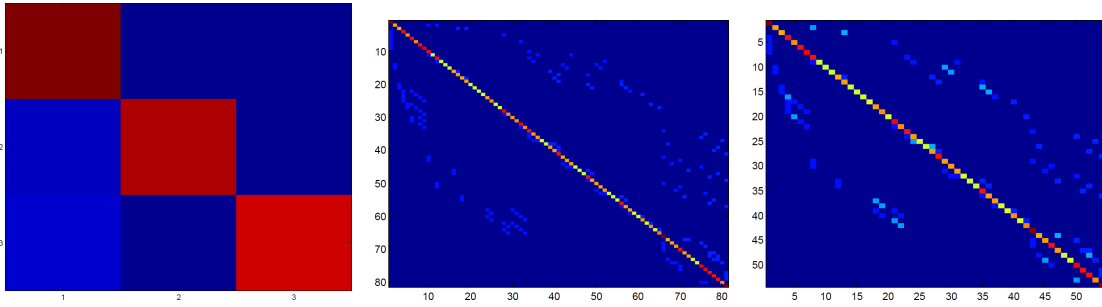
(d) Uncoupled overlap transition matrix $\mathcal{D}_{\pi_4}$

(e) Leaf transition matrix $\mathcal{D}_2$ from the BASS example

(f) Lumped leaf transition matrix $\mathcal{D}'_2$
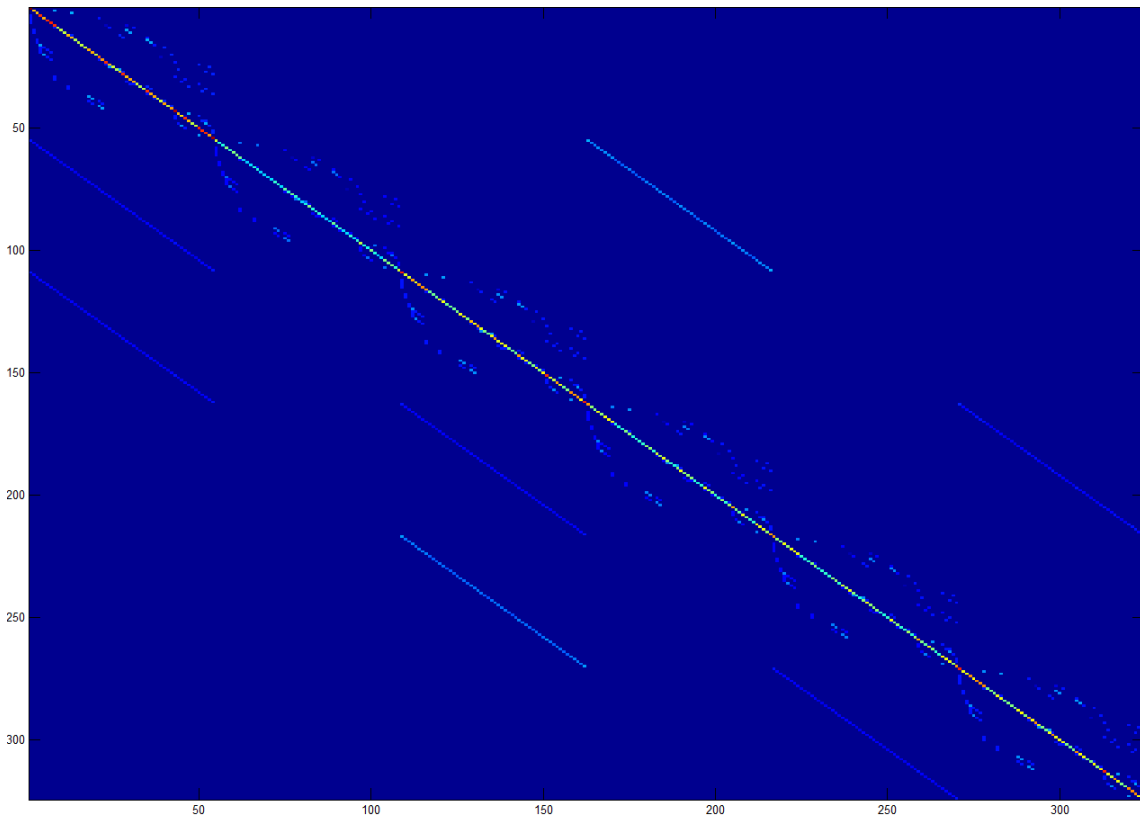
(g) Recomposed lumped product chain $\mathcal{D}'$

Figure A.7: BASS DTMCs

A.5.3 The power grid example

Algorithm A.1 (Composing housing: Sequential interleaving application of the Kronecker product and lumping)**.**

```
1   a=[0.9,0.1;0.2,0.8];
2   matrix = a;
3   counter = 1;
4   numberhouses = 1000;
5   times = [];
6   tic;
7   for i=1:numberhouses-1
8       matrix=kron(a,matrix);
9       %lumping
10      for j=1:counter
11          matrix(:,j+1)=matrix(:,j+1)+matrix(:,j+1+counter);
12      end
13      %deleting superfluous rows and colums
14      matrix_size = size(matrix);
15      for k=1:counter
16          matrix(matrix_size/2+1,:)=[];
17          matrix(:,matrix_size/2+1)=[];
18      end
19      counter=counter+1;
20      store(i)=toc;
21  end
22  csvwrite('matrix.csv',matrix);
```

Algorithm A.2 (Computing limiting window reliability)**.**

```
1   matrix;
2   size_m_temp = size(m);
3   size_m = size_m_temp(1);
4   store = [];
5   [V,D] = eig(m');
6   I=abs(diag(D)-1.)<1e-08;
7   stationary = V(:,1)./sum(V(:,1));
8   for i=1:size_m
9       acc = 0;
10      for j=1:i
11          acc = acc + m(j,size_m-i+j)*stationary(j)+m(size_m-i+j,j)*stationary(size_m-i+j);
12      end
13      store(i) = acc;
14  end
15  store(size_m) = store(size_m)/2;
16  for i=2:size_m
17      store(i) = store(i-1)+store(i);
18  end
19  store = fliplr(store);
20  crashterrain = [];
21  runlength = 100000;
22  for countdown=1:runlength
23      for ct=1:size_m
24          crashterrain(countdown,ct) = 1-((1-store(ct))^countdown);
25      end
26  end
27  imagesc(crashterrain)
28  colormap(hot)
29  colormap(flipud(colormap))
30  colorbar
31  plot2svg
```

A.5.4 The WSN example

Figure A.8 shows the color plots of the corresponding Markov chains.

(a) $\mathcal{D}'_{1,0}$ (b) $\mathcal{D}'_{1,1}$ (c) $\mathcal{D}'_{1,2}$

(d) $\mathcal{D}'_{2,-,0}$ (e) $\mathcal{D}'_{2,-,1}$ (f) $\mathcal{D}'_{2,-,2}$

(g) $\mathcal{D}'_{\text{low}}$ (h) $\mathcal{D}'_{\text{both}}$ (i) $\mathcal{D}'_{\text{up}}$
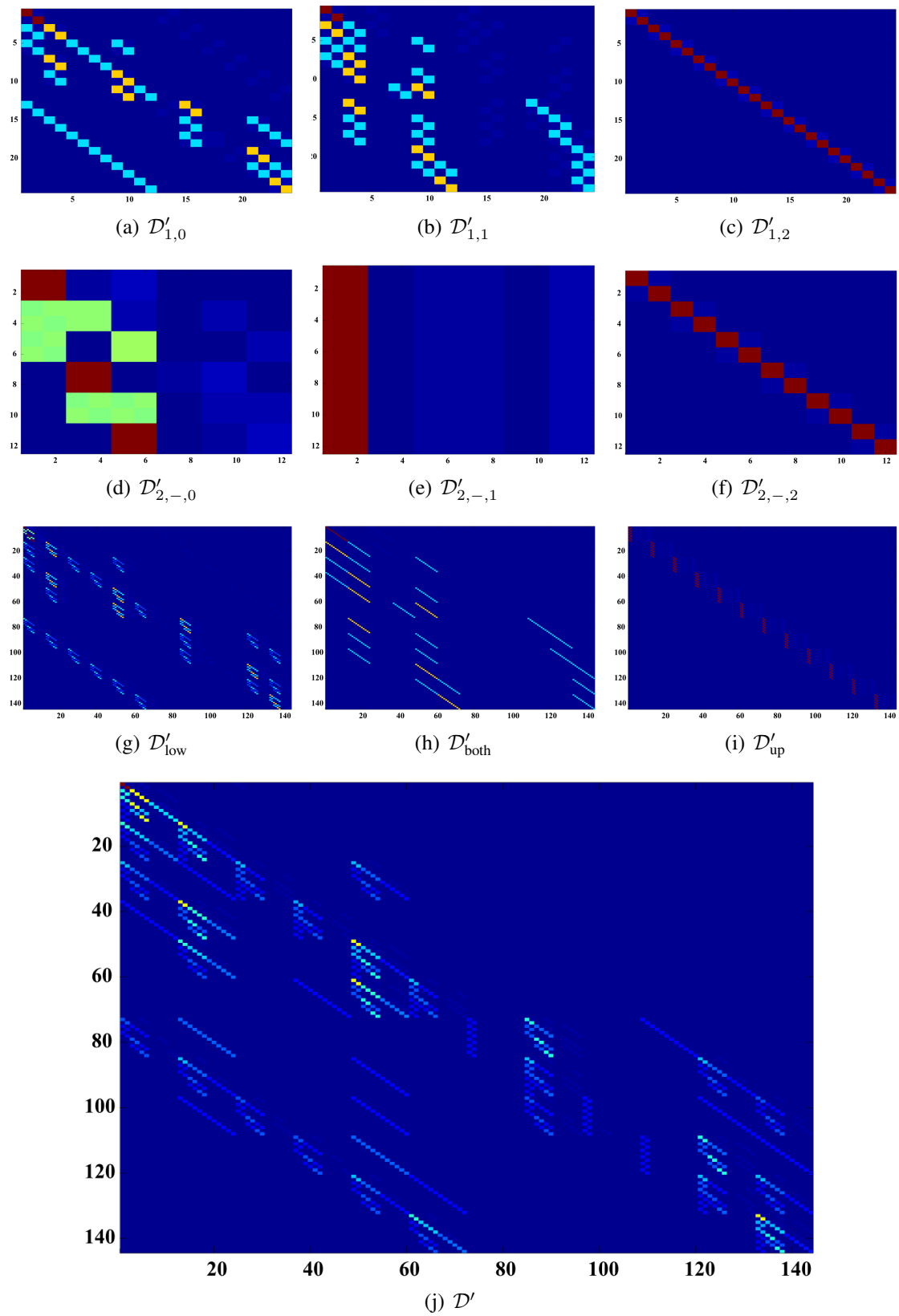
(j) $\mathcal{D}'$

Figure A.8: WSN DTMCs

A.5.5   Counterexample for the double-stroke alphabet

Counterexample

Assume the process topology shown in figure A.9 to execute under a uniformly distributed probabilistic scheduler, a uniformly distributed fault model and the BASS algorithm to be executed on the processes.
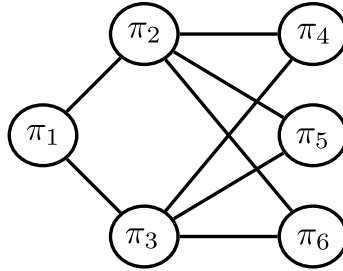

Figure A.9: Example topology showing ambiguity of the double-stroke alphabet

Then, the states

- $\langle 0, 0, 0, 0, 2, 2 \rangle$, $\langle 0, 0, 0, 2, 0, 2 \rangle$ and $\langle 0, 0, 0, 2, 2, 0 \rangle$

belong to one equivalence class[6], and the states

- $\langle 0, 0, 0, 2, 1, 1 \rangle$, $\langle 0, 0, 0, 1, 2, 1 \rangle$ and $\langle 0, 0, 0, 1, 1, 2 \rangle$

belong to another equivalence class. The latter three registers in each class form the coerced register partition. Both equivalence classes claim the label $\langle 0, 0, 0, 4 \rangle$. The example provides one case of ambiguity, showing that the double-stroke alphabet is not universally applicable. Yet, for the examples presented in this thesis, it is applicable and increases the readability.

---

[6] There might be combinations of $\mathfrak{s}$ and p for which $[\langle 0, 0, 0, 0, 2, 2 \rangle]_\sim \sim [\langle 0, 0, 0, 2, 1, 1 \rangle]_\sim$. This counter example accounts for any other case.

A.5.6 MatLab source code: Computing the LWA for the TLA example

Algorithm A.3 (Computing the Stationary Distribution of the TLA - Rounding Errors)**.**

```
1   syms p q real
2   dmatrix = [
3   q+q,q,q,q,q,q,p+q,q,p+q,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
4   q,q+q,q,0,0,q,p+q,0,0,q,0,q,0,0,0,0,0,q,0,p+q,0,0,0,0,0;
5   q,q,q+q,0,0,q,p+q,0,0,0,q,0,q,0,0,0,0,0,q,0,p+q,0,0,0,0,0;
6   q,0,0,q+q,q,0,0,q,p+q,q,q,0,0,q,0,p+q,0,0,0,0,0,0,0,0,0;
7   q,0,0,q,q+q,0,0,q,p+q,0,0,q,q,0,q,0,p+q,0,0,0,0,0,0,0,0;
8   q,q,q,0,0,q+q,p+q,0,0,0,0,0,0,q,q,0,0,0,0,0,0,p+q,q,0,0;
9   q,q,q,0,0,q,p+q+q,0,0,0,0,0,0,0,0,q,p+q,0,0,0,0,0,q,q;
10  q,0,0,q,q,0,0,q+q,p+q,0,0,0,0,0,0,0,0,q,q,0,0,p+q,q,0,0;
11  q,0,0,q,q,0,0,q,p+q+q,0,0,0,0,0,0,0,0,0,0,q,p+q,0,q,0,q;
12  0,q,0,q,0,0,0,0,0,q+q,q,q,0,q,0,p+q,0,q,0,p+q,0,0,0,0,0;
13  0,0,q,q,0,0,0,0,0,q,q+q,0,q,q,0,p+q,0,0,q,0,p+q,0,0,0,0;
14  0,q,0,0,q,0,0,0,0,q,0,q+q,q,0,q,0,p+q,q,0,p+q,0,0,0,0,0;
15  0,0,q,0,q,0,0,0,0,0,q,q,q+q,0,q,0,p+q,0,q,0,p+q,0,0,0,0;
16  0,0,0,q,0,q,0,0,0,q,q,0,0,q+q,q,p+q,0,0,0,0,0,p+q,q,0,0;
17  0,0,0,0,q,q,0,0,0,0,0,q,q,q,q+q,0,p+q,0,0,0,0,p+q,q,0,0;
18  0,0,0,q,0,0,p+q,0,0,q,q,0,0,q,0,p+q+q,q,0,0,0,0,0,0,q,q;
19  0,0,0,0,q,0,q,0,0,0,0,q,q,0,q,q,p+q+q,0,0,0,0,0,0,p+q,q;
20  0,q,0,0,0,0,0,q,0,q,0,q,0,0,0,0,0,q+q,q,p+q,0,p+q,0,q,0;
21  0,0,q,0,0,0,0,q,0,0,q,0,q,0,0,0,0,q,q+q,0,p+q,p+q,0,q,0;
22  0,q,0,0,0,0,0,0,p+q,q,0,q,0,0,0,0,0,q,0,p+q+q,q,0,q,0,q;
23  0,0,q,0,0,0,0,0,q,0,q,0,q,0,0,0,0,0,q,q,p+q+q,0,q,0,p+q;
24  0,0,0,0,0,q,0,q,0,0,0,0,0,q,q,0,0,q,q,0,0,p+q+q,p+q,q,0;
25  0,0,0,0,0,q,0,0,q,0,0,0,0,q,q,0,0,0,0,p+q,q,q,p+q+q,0,q;
26  0,0,0,0,0,0,q,q,0,0,0,0,0,0,0,q,q,q,q,0,0,p+q,0,p+q+q,q;
27  0,0,0,0,0,0,q,0,p+q,0,0,0,0,0,0,p+q,q,0,0,q,q,0,q,q,q+q;
28  ]
29  replaced = subs(dmatrix, {p,q}, {0.375,0.025})
30  [V,D] = eig(replaced')
31  stationary = V(:,1)./sum(V(:,1))
32  vecinit = stationary
33  for i=1:100
34      for j = 1:25
35          store(j,i) = vecinit(j);
36      end
37      vecinit = replaced'*vecinit;
38      vecinit = vecinit ./ sum(vecinit);
39  end
40  vecinit = vecinit'
```

A.5.7 iSat source code: Callaway's TCL example without noise

This section provides the iSat source code that was used to generate figures 7.2 and 7.4 in section 7.1.

Algorithm A.4 (Callaway's TCL in iSat)**.**

```
 1          DECL
 2                  int [0,1] m;                    —thermostat off/on = 0/1
 3                  int [0, 100] h;            —time elapsed
 4                  int [0,50000] acc_load;   —accumulated load
 5                  float [−63,63] theta;     ––(initial) current temperature
 6                  float [−63,63] theta_on;  —deadband lower bound
 7                  float [−63,63] theta_off; —deadband upper bound
 8                  float [0,5000] y;         —accumulated load
 9                  float [0,1] a;            —governs char. thermal mass, a = exp(−h/CR).
                        ↪ with h=0−>a=1
10                  boole theta_gre_theta_off; —Auxiliary variable to enforce (theta<
                        ↪ theta_off) or (theta>=theta_off), avoids non−deterministic
                        ↪ behavior
11                  boole theta_les_theta_on;
12                  define theta_ambient = 32; —ambient temperature (outside)
13                  define theta_set = 20;    —temp. set point
14                  define R = 20;            —thermal resistance C/kW
15                  define P = 14;            —rate of energy transfer kW
16                  define C = 10;            —thermal capacitance kWh/ C
17                  define r = 0.05;
18                  define c = 0.1;
19                  define number_of_houses = 20;
20                  define eta = 2.5;         —load efficiency
21                  define delta = 0.5;       —thermostat deadband
22
23          INIT
24                  —— Conditions at the moment when cooling starts.
25                  m = 0;                          —thermostat is off
26                  h = 1;                    —width of time steps
27                  theta = 26;               —initial room temperature
28                  a = 1;                    —description required
29                  theta_on = theta_set − delta;
30                  theta_off = theta_set + delta;
31                  y = 0;                          —aggregated power demand
32                  acc_load = 0;
33
34          TRANS
35                  theta_on ' = theta_on;
36                  theta_off ' = theta_off;
37                  h' = h;
38                  a' = exp(−h*c*r);               —a depends on time
39                  theta ' = a*theta+(1−a)*(theta_ambient−m*R*P);
40      —the new temperature skipping noise for now: + w;
41
42                  —energy demand
43                  y' = y + P * m;
44                  acc_load ' = acc_load+(number_of_houses*y);
45
46                  —the thermostat switches when it hits the deadband
47                  theta_gre_theta_off <−> (theta > theta_off);
48                  theta_les_theta_on  <−> (theta < theta_on);
49                  theta_gre_theta_off −> (m' = 1);
50                  theta_les_theta_on −> (m' = 0);
51                  (!theta_gre_theta_off and !theta_les_theta_on) −> m'=m;
52
53          TARGET
54                  (h = 100) and (y > 500);
```

A.6 Curriculum vitæ

Personal data

First Name	**Nils**
Middle Name	**Henning**
Family Name	**Müllner**
Date of Birth	November 19 1979
Place of Birth	Bremen, Germany

Current activities

since August 1 2013	**Research assistant** Carl von Ossietzky Universität Oldenburg, Germany, Department of Computer Science, Interdisciplinary Research Center on Critical Systems Engineering for Socio-Technical Systems, Supervisors: Prof. Dr. Martin Fränzle, PD Dr. Sibylle Fröschle
since April 1 2008	**PhD student**, Carl von Ossietzky Universität Oldenburg, Germany, Department of Computer Science, Graduate School TrustSoft, System software and Distributed Systems Group, Supervisor: Prof. Dr.-Ing. Oliver Theel

Studies

2007 − 2014	**Promotion Informatik**, Carl von Ossietzky Universität Oldenburg, Germany, PhD Thesis: Unmasking fault tolerance – Quantifying deterministic recovery dynamics in probabilistic environments.
2000 − 2007	**Diplom Informatik**, Carl von Ossietzky Universität Oldenburg, Germany, Diploma thesis: Simulation of Self-Stabilizing Distributed Algorithms to Determine Fault Tolerance Measures [Müllner, 2007], Supervisor: Prof. Dr.-Ing. Oliver Theel

School

1990 − 1999	**Gymnasium (High school)**, Ökumenisches Gymnasium, Bremen, Germany. Passed with Abitur.
1986 − 1990	**Grundschule (Elementary school)**, Grundschule Oberneuland, Bremen, Germany

Work experience

since 2013	**Research assistant** Carl von Ossietzky Universität, Oldenburg, Department of Computer Science, CSE: Interdisciplinary Research Center on Critical Systems Engineering for Socio-Technical Systems
$2011 - 2013$	**Research assistant** OFFIS Institute for Computer Science, MoVeS: *Mo*deling, *Ve*rification, and control of complex *S*ystems, Funded by the European Commission, ICT Research in FP7-ICT-2009-257005, Supervisor: Prof. Dr. Martin Fränzle
$2008 - 2011$	**Scholarship holder** Carl von Ossietzky Universität, Oldenburg, Department of Computer Science, TrustSoft Graduate College, funded by the German Research Council (DFG), Supervisors: Prof. Dr.-Ing. Oliver Theel, Prof. Dr. Ernst-Rüdiger Olderog, Prof. Dr. Martin Fränzle.
$2007 - 2008$	**Research assistant** Carl von Ossietzky Universität, Oldenburg, Department of Computer Science, AVACS SFB/TR 14 (*A*utomatic *Ve*rification and *A*nalysis of *C*omplex *S*ystems, Transregional Collaborative Research Center), funded by the German Research Council (DFG), Supervisor: Prof. Dr.-Ing. Oliver Theel.
$2002 - 2003$	**Student assistant** Berufsakademie Oldenburg (University of Cooperative Education).

Language Skills

German	**Native speaker**
English	**Excellent** first foreign language, PhD studies in English, Cambridge First Certificate
Latin	**Basics** second foreign language in school
Spanish	**Basics** third foreign language in school

Honors

Individual Project	Ausgezeichnet (Distinguished) (awarded for $\geq 95\%$), Supervisor: PD Dr. Elke Wilkeit, Dr. Hans Fleischhack, Topic: Logisch-Funktionale Sprachen im Vergleich (Comparison of Logical-Functional Languages). The individual project in the former German "Diplomstudiengang" coincides with the bachelor thesis today.
TrustSoft	Full scholarship awarded for three years within the TrustSoft Graduate College by the German Research Council (DFG)
AINA2012	Best Paper Award, best of 361 published papers, first author, 29% acceptance rate on main track

Academic Services

FGCS 2014	Future Generation Computer Systems, Springer Journal
DPNoS 2014	The 2014 International Workshop on the Design and Performance of Networks on Chip
ICEPIT 2014	The 2014 International Conference on Electronic Publishing and Information Technology
IJCDS'V3	International Journal of Computing and Digital Systems
CSS 2013	The 5th International Symposium on Cyberspace Safety and Security (CSS 2013)
ADC 2013	The 24th Australasian Database Conference
ICDKE 2012	2012 International Conference on Data and Knowledge Engineering
SEFM 2012	The Tenth International Conference on Software Engineering and Formal Methods reviewed for Martin Fränzle
EMSOFT 2012	The Twelfth International Conference on Embedded Software reviewed for Martin Fränzle
ICDKE 2011	2011 International Conference on Data and Knowledge Engineering
ARES 2010	The Fifth International Conference on Availability, Reliability and Security reviewed for Oliver Theel