Formal Configuration of Fault-Tolerant Systems

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von

Linda Herrmann geb. Leuschner
geboren am 14. Oktober 1988 in Dresden

Betreuende Hochschullehrerin
Prof. Dr. rer. nat. Christel Baier

Dresden, 21. Dezember 2018

Abstract

Bit flips are known to be a source of strange system behavior, failures, and crashes. They can cause dramatic financial loss, security breaches, or even harm human life. Caused by energized particles arising from, e.g., cosmic rays or heat, they are hardly avoidable. Due to transistor sizes becoming smaller and smaller, modern hardware becomes more and more prone to bit flips. This yields a high scientific interest, and many techniques to make systems more resilient against bit flips are developed. Fault-tolerance techniques are techniques that detect and react to bit flips or their effects. Before using these techniques, they typically need to be configured for the particular system they shall protect, the grade of resilience that shall be achieved, and the environment. State-of-the-art configuration approaches have a high risk of being imprecise, of being affected by undesired side effects, and of yielding questionable resilience measures.

In this thesis we encourage the usage of formal methods for resiliency configuration, point out advantages and investigate difficulties. We exemplarily investigate two systems that are equipped with fault-tolerance techniques, and we apply parametric variants of probabilistic model checking to obtain optimal configurations for pre-defined resilience criteria. Probabilistic model checking is an automated formal method that operates on Markov models, i.e., state-based models with probabilistic transitions, where costs or rewards can be assigned to states and transitions. Probabilistic model checking can be used to compute, e.g., the probability of having a failure, the conditional probability of detecting an error in case of bit-flip occurrence, or the overhead that arises due to error detection and correction. Parametric variants of probabilistic model checking allow parameters in the transition probabilities and in the costs and rewards. Instead of computing values for probabilities and overhead, parametric variants compute rational functions. These functions can then be analyzed for optimality.

The considered fault-tolerant systems are inspired by the work of project partners. The first system is an inter-process communication protocol as it is used in the Fiasco.OC microkernel. The communication structures provided by the kernel are protected against bit flips by a fault-tolerance technique. The second system is inspired by the redo-based fault-tolerance technique HAFT. This technique protects an application against bit flips by partitioning the application's instruction flow into transaction, adding redundance, and redoing single transactions in case of error detection.

Driven by these examples, we study challenges when using probabilistic model checking for fault-tolerance configuration and present solutions. We show that small transition probabilities, as they arise in error models, can be a cause of previously

known accuracy issues, when using numeric solver in probabilistic model checking. We argue that the use of non-iterative methods is an acceptable alternative. We debate on the usability of the rational functions for finding optimal configurations, and show that for relatively short rational functions the usage of mathematical methods is appropriate.

The redo-based fault-tolerance model suffers from the well-known state-explosion problem. We present a new technique, counter-based factorization, that tackles this problem for system models that do not scale because of a counter, as it is the case for this fault-tolerance model. This technique utilizes the chain-like structure that arises from the counter, splits the model into several parts, and computes local characteristics (in terms of rational functions) for these parts. These local characteristics can then be combined to retrieve global resiliency and overhead measures. The rational functions retrieved for the redo-based fault-tolerance model are huge — for small model instances they already have the size of more than one gigabyte. We therefor can not apply precise mathematic methods to these functions. Instead, we use the short, matrix-based representation, that arises from factorization, to point-wise evaluate the functions. Using this approach, we systematically explore the design space of the redo-based fault-tolerance model and retrieve sweet-spot configurations.

Danksagung

Zuallererst möchte ich meiner Betreuerin, Christel Baier, herzlich danken. Sie nahm sich viel Zeit für wissenschaftliche Diskussionen und hatte immer einen guten Rat, wenn ich feststeckte. Ohne sie wäre diese Arbeit niemals entstanden. Ich möchte auch Christof Fetzer und Hermann Härtig dafür danken, dass sie unzählige (manchmal vermutlich ziemlich einfältige) Fragen darüber beantwortet haben, wie Rechner denn nun eigentlich funktionieren, wenn man sie aus einer nicht-theoretischen Perspektive betrachtet.

Ein großes Dankeschön geht an meine Kollegen. In vielen wissenschaftlichen und nicht-wissenschaftlichen Diskussionen erörterten wir gemeinsam die Abstrusitäten dieser Welt. Diese Zeit mit euch zu verbringen war wundervoll, ich werde diese tollen Stunden nie vergessen. Besonders erwähnen möchte ich Karina und Sandy, welche mir in der Wust an organisatorischen Dingen immer hilfsbereit zur Seite standen, Sascha, welcher immer ein offenes Ohr für die kleinen und großen Probleme meiner Publikationen und dieser Arbeit hatte, Joachim, welcher mir oft half, meinem Rechner meinen Willen aufzuzwingen, und David, welcher mit interessanten Diskussionen über Philosophie, Juristerei und Medizin den Arbeitsalltag beträchtlich auflockerte.

Viele schöne Stunden verbrachte ich mit den Freunden der Mimimi-Bouldergruppe. Gemeinsam sorgten wir beim Klettern und Bouldern für den körperlichen Ausgleich, hatten aber auch beim Zocken, Karaoke und bei Spieleabenden gemeinsam viel Spaß. Beim nun schon traditionellen Fehlertrinken fanden sie noch ca. 150 Fehler und Formulierungsschwächen in dieser Arbeit.

Zuletzt möchte ich den drei wichtigsten Menschen an meiner Seite danken. Danke, Mutti und Vati, dass ihr Zeit meines Lebens immer ein offenes Ohr für meine Probleme hattet und mich in meinen Entscheidungen unterstützt habt. Danke, Kai, für die vielen tollen Sachen die wir gemeinsam machen, und dafür, dass du mich immer, selbst in schweren und stressigen Zeiten, zum Lachen bringst, mir Mut machst und Kraft gibst. Bevor ich dich traf, hätte ich nie geglaubt, dass es einen Menschen gibt, der mir so wichtig ist wie du.

Contents

1 Introduction **7**

1.1 Bit Flips and Fault Tolerance 7
1.2 Configuration of Fault-Tolerance Techniques 9
1.3 Fault-Tolerant Inter-Process Communication 12
1.4 Redo-Based Fault Tolerance 14
1.5 Counter-Based Factorization 15
1.6 Contribution 16
1.7 Related Work 16
1.8 Outline 19
1.9 Own Publications 19
1.10 Resources 19

2 Preliminaries **21**

3 Configuration of Fault-Tolerant Inter-Process Communication **25**

3.1 Inter-Process Communication in Fiasco.OC 26
3.2 Modeling Fault-Tolerant Inter-Process Communication 31
3.3 The Space-Probe Example 36
3.4 Time Consumption and Accuracy of Iterative PMC Methods 38
3.5 Configuration of Resilient Communicating Processes 43

4 Counter-based Factorization **51**

4.1 Base Schema of Factorization 52
4.2 Counters 56
4.3 Factorization for Simple Counters 61
4.4 Implementation 68

5 Configuration of Redo-Based Fault-Tolerance Protocols **77**

5.1 The Concept of Redo-Based Fault Tolerance 78
5.2 A Model for Redo-Based Fault Tolerance 80
5.3 Configuration of Redo-Based Fault Tolerance 88

6 Conclusion and Further Work **107**

Bibliography **109**

1 Introduction

Scaling trends for modern hardware cause transistors to become smaller and smaller. Despite the undisputed advantage that these trends result in more computational power, they also have the effect of hardware becoming more prone to bit flips [Shi+02; Sha+14]. These bit flips are known to be a source of system failures [SG10; NDO11; Gun+14]. To increase the resilience of systems against bit flips, fault-tolerance techniques are applied.

The driving goal of this thesis is to study the configuration of fault-tolerance techniques via formal methods. The configuration goal is to guarantee a predefined level of resilience against bit flips with acceptable overhead, with respect to the underlying system and the environment.

We exemplarily configure the system variables of two concrete systems. The first system is a system of interacting processes that communicate via inter-process communication structures provided by the L4/Fiasco.OC microkernel [16b]. The other one is an application-level fault-tolerance mechanism that performs error correction by redoing parts of the application. The mechanism is inspired by the redo-based fault-tolerance technique HAFT [Kuv+16]. These systems arose from the work of project partners in the "Center for Advancing Electronics Dresden" (cfaed). The goal of cfaed is the exploration of new technologies for electronic information processing.

The exemplary configuration is performed with formal methods, in particular probabilistic model checking. The usage of formal methods comes with several benefits, but also with challenges. In this thesis, we show how to overcome these challenges and how to apply formal methods to configure systems with respect to resilience criteria.

1.1 Bit Flips and Fault Tolerance

Bit Flips. In modern hardware, transistors have a size of some nanometers, with sizes still shrinking. In 2016 industrial production of chips following the 10 nanometers technology[1] started [Sam16], mass production of chips of 7 nanometer technology began in June 2018 [Wei18]. Industrial production of 5 nanometer technology is expected for 2020 [TSM18]. With the size of a transistor also the power the transistor needs to switch from the "off" to the "on" state decreases. Reducing this power has the drawback of making modern transistors more vulnerable to uncontrolled energy impacts [Shi+02]. Energized particles that carry more energy than needed to switch

---

[1] x nanometer technology refers to transistors with a gate length of x nanometers.

a transistor's state, can cause the state of the transistor to change unintentionally [Li+04]. These energized particles are, e.g., heavy ions, neutrons, or photons, caused by, e.g., cosmic radiation [McK+96], alpha particles from packaging materials [MW79], or interaction of cosmic ray thermal neutrons with the ^{10}B isotope of Boron [Bau+95]. On earth, occurrence of many high-energized particles has been reduced with better process technologies and removing uranium and thorium impurities [Bau05]. But the scale of modern transistors makes them vulnerable to particles carrying much less energy, e.g., particles caused by radiation. [Zie98] shows that these particles are much more frequent in atmosphere than high-energized particles. In outer space, high energized particles (e.g. from solar storms), are still present at a high rate [Sup14] and give an additional chance of unintentional changes of modern transistors' state.

The change of a transistor's state is called a *bit flip*, *soft fault*, *transient faults* or *single event upset*. In this thesis, we will use the term "bit flip". The rate of bit flips is typically measured in failures in time (FIT), and describes the number of bit flips that arise in one megabite of memory within 10^9 hours of operations. The concrete rate of bit flips depends on the hardware and the environment. [SL12] gives an on-earth bit-flip rate of 0.066 FIT. Other estimates give bit-flip rates of 0.061 FIT [Li+10] and 0.044 FIT [Sri+13]. This means that in a single operation of a 2.5GHz processor an error occurs approximately with a probability of 10^{-15}.

The effects of bit flips are manifold. They might have no effect at all (e.g., if corrupted bits are refreshed before they are read), but they also can cause applications or systems to crash or produce wrong results. The latter is of special interest, since wrong data can cause security breaches, financial loss, or death. In 2014, [Kim+14] showed that in modern caches repeatedly refreshing a cash line causes adjacent cache lines to be affected by bit flips with a high chance. [SD15] extended this work and showed that this effect can be used to maliciously obtain kernel privileges. This technique is known as "rowhammer". In 2007, an uncontrolledly accelerating car caused the death of a woman [EET13a]. In this car, the process measuring the angle of the throttle control was activated and deactivated by a single, unprotected bit. A bit flip in this bit caused the process to deactivate while the throttle control was depressed and thus the throttle to be "software-stucked" at accelerating position [EET13b]. These incidents illustrate the need of protection against bit flips.

Fault-Tolerance Techniques. Using the terminology of [Sah05], we say that a bit flip, independent of its effect, is a *fault*. If the bit flip affects the run of the system, the system has an *error*. These errors can be detected and corrected to avoid *failures*, i.e., undesired system behavior. The concrete definition of a failure hereby depends on the system. For systems providing a high availability like airplane or stock market software, a crash of single processes or even the whole system is a failure. However, in an operating system for home computers, a failure is typically defined as unintended and unobservable data changes or access granting, i.e., a silent data corruption. A crash of a process is, from this operating system's perspective, detectable and repairable, and thus an error.

To avoid system failures, fault-tolerance techniques are applied. The *resilience* (also *resiliency*) of a system is a measure for its ability to deal with errors. Fault-tolerance techniques equip a system with redundance and use this redundance to detect and correct errors. They increase the system's resilience and thus avoid failures. The number of existing fault-tolerance techniques is enormous. Each one is tailored to the system it shall protect (e.g., air planes, hospital software, home computers), to the architecture level (e.g., hardware, operating system, user applications), to the environment (e.g., high or low bit-flip rates), to characteristics of the concrete application scenario (e.g., multi-threaded or single-threaded), and to many more dimensions. Furthermore, fault-tolerance techniques can be tailored to particular hardware or operating systems (e.g., Intel cores or Mac OS file systems), some of them require user interaction, and they differ in the type and number of bit flips they can find and correct. But all of them have one thing in common: They cause overhead. Adding and maintaining redundance causes overhead in terms of time and energy. For many fault-tolerance techniques this opens a configuration problem: How much redundance shall be added, to provide a certain level of resilience without paying to much? How often or at which points in time shall error detection be performed? Is there a sweet-spot configuration that reduces overhead without decreasing resilience?

The answer to these questions typically depends not only on the concrete system and resilience/overhead requirements, but also on the environment. As an example, consider a fault-tolerance mechanism that shall protect high-level applications against up to two bit flips arising at the same time. For this purpose, it periodically checks the application's data and, if it detects an error, performs error correction. If, between two checks, three bit flips arise in the application's data, the fault-tolerance mechanism might not be able to detect these three bit flips. In this case, there is a failure in the system and will cause data loss or a crash eventually. The time between two periodic checks shall now be configured such that the probability of a failure is below 0.0001%, while having as less overhead as possible. Performing checks too seldom will cause the chance of a failure to be too high, performing checks too often will cause additional, unnecessary overhead due to error detection. The optimal check interval depends on the applications characteristic (e.g., on the amount of data that shall be protected), on the hardware (modern hardware is more prone to bit flips), and on the environment (the probability of bit flips is higher in high altitudes or in space). This makes the configuration a difficult task.

1.2 Configuration of Fault-Tolerance Techniques

State-of-the-Art Configuration. Most fault-tolerance techniques are configured to their use case using one of the state-of-the-art analysis methods: simulation and fault injection. During simulation, an erroneous environment is simulated and the error-prone system, equipped with the fault-tolerance technique, is run in this simulated environment. Fault injection in contrast is a method that assumes a bit-flip-free environment, and injects errors in the system to analyze their effects. For configuration, the simulation-based or fault-injection-based analysis is performed for a set of possible configurations, and the best (with respect to some optimality criteria) is chosen.

Both methods have critical drawbacks. Simulating an erroneous environment results in uncontrollable, imprecise, and probably fluctuant error occurrence. Typically, there is no control about the amount, type, and target of bit flips. Furthermore, although effects of failures can be very drastic, the source of errors, bit flips, is a rare event. Simulations of a fault-tolerant system in an environment with realistic bit-flip rates, that shall provide reliable results, would need a very long runtime.

The injection of faults in contrast can be done very precisely. Yet, a complete analysis of fault effects would require to perform fault injection very often: A fault needs to be injected at every point in time in every memory cell. Therefore, a common fault injection technique is *randomized fault injection*, where a probabilistically chosen subset of all possible faults is injected. Unfortunately, this technique's measurement results are independent of the bit-flip rate. Resilience measurements retrieved from fault injection typically are of the form "out of n injected errors, x were detected, y were corrected, and z led to failures". [SBS15] showed that this kind of analysis can be manipulated easily. As an example, they applied (randomized and complete) fault injection to evaluate a fault-tolerance mechanism that does not detect any error, but just consumes time and memory. They showed that with the standard fault-injection resilience measurements applying this useless fault-tolerance mechanism increases resilience.

As another drawback of fault injection, injected faults can be "canceled" due to the complexity of modern hardware. For example, injecting a fault in the cache does not result in an error, if the cache line is marked as invalid and thus the respective data is reloaded into the cache. Then, the injected fault is just annulated.

Both techniques, simulation and probabilistic fault injection, are of stochastic nature and thus can not guarantee correct results. Furthermore, the techniques typically are applied to a benchmark of concrete systems, and thus the results are tailored to these selected systems. There is no evidence that the results are generalizable. The benchmark systems are often black boxes, which makes retrieving more complex resilience criteria difficult. Increasing the observability means changing the system. Then, a different system's resilience is analyzed, and it is not clear whether results are valid for the original system. Finally, since the analysis method needs to be applied once for each considered configuration, using these state-of-the-art techniques is very time-consuming.

Configuration with Formal Methods. The variety of formal methods is very broad. There are pen-and-paper methods, which are often efficient methods, e.g., in the field of coding theory [Sha48b; Lin98; AHS12]. Theorem proving [BC10; Gal15] is a semi-automated formal method that can be used to deduce formulas describing system characteristics. Statistical hypotheses testing [LR05], as it is used, e.g., in stochastic model checking [YS02; KNP07], is a simulation-based formal method that combines benefits of simulation techniques with formal methods' benefits.

An automated formal method that can address all the mentioned problems is probabilistic model checking [EC80; Var85; BK08] (PMC). PMC is a widely used Markov-model-based technique. A Markov model is a state-based model with probabilistic transitions. States can be annotated with costs or rewards. In a model for a resilient system, states in the Markov model represent system and error states, transitions model system behavior and error occurrence, and costs/reward annotations mark, e.g., the overhead arising through error detection and correction. The model's level of detail determines the observability of the modeled system and the complexity of the resilience criteria that can be chosen for configuration. This also means that the abstraction level can be chosen such that relevant system criteria are modeled while not modeling a concrete system and thus configuring a fault-tolerance technique for a set of concrete systems. Error occurrence and effects can be controlled completely by simply modeling the desired behavior, and side effects can be excluded from the model.

PMC is a precise method, i.e., results are guaranteed to be correct. In this work we will utilize variants of PMC that allow parameters in the underlying Markov model, and compute rational functions instead of single values. Although these variants are restricted to parametrized transition probabilities and rewards, this eases the configuration process. By using system variables as parameters, PMC does not need to be invoked for each considered configuration, instead, a rational function describing the resilience or overhead of the system is obtained and can be analyzed for optimality. In this thesis, when not stated explicitly otherwise, we always use these parametric variants of PMC, since we always consider parametric models.

An effect that is typically only vaguely included in state-of-the-art configurations, is the error-proneness of fault-tolerance techniques itself. As stated before, resilience is improved by adding redundance to the error-prone system. This redundance itself can be affected by errors. Effects of errors in the fault-tolerance techniques can be easily included in the model, when using formal methods, and thus make the configuration holistic.

There is no free lunch. PMC, despite its advantages, is often difficult to apply. One obstacle is the modeling process itself. Finding the right level of abstraction is difficult, and modeling a system by hand is error-prone. For large models, time and memory consumption during PMC can be another show-stopper. The purpose of this work is to demonstrate how these difficulties can be handled in the field of fault-tolerance configuration with respect to resilience criteria.

Probabilistic model checkers like Storm [Deh+17] or Prism [KNP11] implement PMC techniques. Both Storm and Prism also implement variants for parametric models. In this work, we use Storm to apply PMC.

1.3 Fault-Tolerant Inter-Process Communication

Inter-process communication (IPC) is an operating system service that handles exchange of data of several processes forming a system or a part of a system. The communicating processes hereby shall be anonymous in the sense that a process does not need to know the address space of its communication partner. Yet, exchanged data shall be transferred to the correct receiver, without data loss or enabling access to this data for other processes than the communicating ones. The operating system provides communication structures that need to be protected against bit flips to guarantee trouble-free communication.

Inter-Process Communication in Fiasco.OC. In this thesis we consider IPC in the L4/Fiasco.OC microkernel [16b]. Here, communication is controlled by the kernel completely. The kernel provides IPC channels, grants processes read or write access to these channels, and maintains information on authorized processes. A process A does not send a message directly to process B but uses a channel c connecting A and B. For this purpose it performs a system call, informing the kernel that it needs to send a message via channel c. The operating system checks whether A and B are authorized to send and receive messages via this channel and copies the message from A's memory to B's memory.

The communication structures provided by the operating system comprise, e.g., the communication channels and data describing which processes are connected to these channels. Errors in the communication structures affect, e.g., data storing the receiving process' ID. This error might be detectable, e.g., if the information is falsified to some useless data. In this case, the operating system will throw an error, which typically causes an exception in one of the processes. This exception can be caught and handled, and thus the described scenario is not a failure. In contrast, errors may concern communication structures such that they are not detectable and, e.g., falsify the information such that, instead of process B being connected to channel c, a process F is connected. In this case, F receives messages it is not intended to have, which is a security breach or can cause unintended behavior of process F. A failure arose during communication.

Both detectable errors and failures affect the utility of the system. The concrete definition of utility depends on the system and its purpose. In Chapter 3, we will demonstrate how a fault-tolerance technique can be configured such that the utility is maximized.

The contribution of this part of the thesis is two-fold. First, we use an exemplary set of communicating processes to reveal problems with iterative standard PMC methods. Iterative methods can not handle parametric models, yet they are typically much faster than non-iterative methods. It is well known that the most common iterative method, value iteration [Bel57], can not guarantee accuracy bounds for the result [HM18]. We show that, arising from small transition probabilities modeling error occurrence, this problem of inaccuracy is relevant for resilience analysis. In fact, with the most common (non-parametric) PMC method we retrieve probability results that are off by about 50%. Furthermore, we show that the proposed solution interval iteration [HM18] solves the accuracy problem, but costs a lot of time. We reveal that for the exemplary model the usage of non-iterative methods is the best choice. Since this insight is based on the tiny transition probabilities, we argue that during resiliency analysis and configuration iterative methods should always be handled with care. We then show how to use non-iterative methods for parametric models to configure inter-process-communication systems with respect to resilience criteria.

Modeling and Configuration. Modeling the system involves modeling three parts: the functionality of inter-process communication in Fiasco.OC, the error model, including effects of errors, and the communication structure of the processes in the considered system. The latter refers to the relevant characteristics of the participating processes (e.g., working time between two communications) and information on which processes can communicate with each other. To ease the modeling process, we provide a domain specific language to describe the process characteristics and communication structure. A model generation tool then generates the model from this description, and automatically includes IPC functionality, error effects, and additional information about the error model.

We exemplify the use of the generation tool and the configuration procedure on a concrete set of communicating processes forming a part of a space probe. This example is inspired by the Maven Mars probe [Jak+15; 11a]. The modeled part is responsible for collecting data in space and transferring it to earth. Bit flips in the communication structures can, if not detected and corrected, cause message loss or strange system behavior. A safe way to react to detected bit flips in the communication structures is to restart the system. The probability of detecting an error depends on the chosen fault-tolerance mechanism. After a restart, all memory is refreshed and the system is error-free again. To handle undetected bit flips, the space probe is restarted in regular time intervals, independent of the current error state. As a worst-case assumption we say that in case of an undetected bit flip the system is not working correctly anymore, as soon as the error arose. The utility of this system is defined as the availability, i.e., the time the system does not suffer from an undetected error and is not in a restart progress. The configuration goal is to set the interval length between two error-independent restarts such that the availability is maximized. Restarting too often will cause availability loss due to unnecessary restarts. Restarting too seldom will cause availability loss due to undetected errors remaining too long in the system. The best interval length depends on the error detection probability. We consider a set of fixed probabilities and configure the space probe with respect to each of these probabilities. In [Leu+17], we configured a fixed restart time interval, i.e., we found the best t such that between two error-independent restarts exactly t microseconds elapsed. In [Her+18b] and in this thesis, we find the optimal expected interval length, i.e., in each time step there is a chance p of performing a restart. The probabilistic restart models the use of heuristics when deciding whether to perform a restart or not. These heuristics may consider, e.g., the overall observable state of the space probe, the space weather (which can increase bit-flip probabilities and thus the chance of having undetected errors), etc. The retrieved optimal value for p can then be used for the further configuration of the heuristics.

We apply PMC on the space probe model, and use p and the detection probability as parameters, and obtain a rational function. For each selected detection probability, we appoint this probability in the rational function and apply Newton's method to retrieve the optimal value for p. After finding optimal configurations (one for each chosen detection probability) we analyze the space probe in this optimal settings. An interesting result of this analysis is that the optimal configuration involves about 99% of all restarts to be performed although the system is error-free.

1.4 Redo-Based Fault Tolerance

The other fault-tolerance technique considered in this thesis, redo-based fault tolerance, protects a sequence of instructions and the produced data against bit flips. This instruction sequence can be, e.g., a program, an application, or an operating system service. In the further, we call the instruction sequence *application* and assume the sequence to consist of finitely many instructions. The instruction sequence and the data is equipped with redundance, and both the original instructions and the data as well as the redundance is prone to bit flips. The application is partitioned into *transactions*. During execution, each transaction is checked for errors. If an error is detected, the transaction is redone. During this redo, the executed instructions and the produced data can again be affected by bit flips and cause another redo. After a predefined number of redos the application is aborted instead of performing another redo. If no error is detected, the next transaction is executed. If there was an undetected error, i.e., a failure, in some transaction and thus the next transaction is executed, this failure will remain in the application until all instructions are executed or the application is aborted due to another error. After executing all transaction, the application terminates. We are interested in tuning the probabilities of the three possible outcomes: application abort, application termination with a failure, and correct application termination.

Modeling and Configuration. The model for redo-based fault-tolerance models a general application protected by a general redo-based fault-tolerance mechanism. It contains many attributes, which can be set to concrete values to obtain an application with concrete characteristics and a concrete redo-based fault-tolerance mechanism. For example, the length of the application and the overhead caused by error detection can be defined by setting the respective attributes.

In the exemplary configuration we choose a concrete redo-based fault-tolerance mechanism that is inspired by HAFT [Kuv+16], and an application that consists of 10^{12} instructions. We configure the following system variables:

- The number of instructions in a transaction.

- The maximal number of redos that can be performed before aborting the application.
- The amount of redundance that shall be added.

The system shall be configured such that the probability of terminating correctly is above 99.95% while the conditional probability of terminating with a failure in case of not terminating correctly is below 85%. From all configurations satisfying these conditions, we choose the one with minimal overhead.

The amount of redundance directly correlates with the detection probability and thus can be treated as parameter. Neither the number of instructions per transaction nor the maximal number of redos can be treated as parameter, when using standard PMC techniques. We configure the redo-based fault-tolerance technique for several error probabilities. To not invoke probabilistic model checking for each error probability, we handle the error probability as additional parameter and appoint concrete values later, after retrieving the rational functions from parametric model checking. However, the parametric model of an application that consists of 10^{12} instructions and is equipped with a fault-tolerance technique is ways too large to perform standard PMC. Depending on the number of instructions per transaction, it has up to 10^{10} states. Therefore, we developed and present a new factorization technique that makes use of the model structure. An overview of this technique will be provided in the next section. The new technique can handle the huge state space and thus tackles the state-explosion problem arising in this model. Furthermore, it allows to treat the number of instructions per transaction parametric.

In contrast to the IPC configuration, where the retrieved rational functions are small enough to perform a mathematical analysis, the rational functions for this protocol are huge (in the gigabyte range). For this reason we now fix a set of configurations and choose the best from this set. We show how to systematically explore the set of configurations, by first determining the best number of maximal redos, then fixing this setting and exploring the remaining configurations. We retrieve the unexpected result that for the chosen error probabilities it is always advisable to perform at most one redo.

1.5 Counter-Based Factorization

In the redo-based fault-tolerance model, the sub-model of the execution of one transaction, error detection and redos consists of only several hundred states. A counter counts the number of transactions that are already performed. When the counter reaches its maximal value the application terminates. For the example application consisting of 10^{12} instructions, the counters maximal value is, depending on the chosen configuration, up to 10^{10}, and thus causes the overall model to be very large. For this reason, standard PMC methods can not compute rational functions for resilience properties of the model, despite the small size of single-transaction sub-models.

The transaction counter induces a chain-like model structure. In this thesis, we present a new technique, called counter-based factorization. Counter-based factorization is a mathematical approach that uses this structure and splits the model into chain links, called factors. Each factor is analyzed to retrieve local resilience and overhead characteristics (in terms of rational functions), and the local results are composed to obtain global characteristics, i.e., rational functions that can be used for configuration. The approach is not tailored to the redo-based fault-tolerance setting, but generalized to arbitrary discrete-time Markov chains where a chain-like structure arises from a counter, i.e., a variable whose value never decreases. We characterize counters in terms of being simple (the counter value is always increased by the same value) and observing (the counter value does not influence other modeled behavior), and present specialized instances of the counter-based factorization approach.

The transaction counter in the redo-based fault-tolerance model is a simple, observing counter. We present `fact`, a python tool that implements the factorization approach for this particular type of counters. It enables counter-based factorization for the redo-based fault-tolerance protocol, and can also be applied to arbitrary models having a similar structure. `fact` combines methods from PMC and computer algebra systems. PMC is utilized to retrieve local characteristics, and the computer algebra system is used to deduce global rational functions from the local characteristics. The probabilistic model checker used by `fact` is Storm. As a computer algebra system, we use the python-library sympy.

1.6 Contribution

The main contribution of this work is the investigation of the adequateness of formal methods for fault-tolerance configuration with respect to resilience-overhead constraints. The exemplary configuration of the two particular fault-tolerance techniques with PMC shows this applicability, examines difficulties when applying PMC, and presents solutions. A sub-contribution is the counter-based factorization framework and the implementation of its special case for simple, observing counters in `fact`.

1.7 Related Work

Formal methods have widely been used to configure systems of all kinds. For example, Fränzle et al. [Frä+15] present an approach that combines model checking, importance sampling, and SAT-modulo-theory solving to configure hybrid systems. They apply this approach to obtain a configuration for a battery charger such that the battery is sufficiently charged at sunset within 90% of all days. In [LR16], Long and Rinard use machine learning to automatically generate patches. [Che+13] combine Monte Carlo sampling and the swarm intelligence method to repair an adaptive system online. In the field of controller syntheses, optimal schedules for real-time tasks are synthesized in [Abd+18; Jia+17; AM01]. [AMY17] synthesize Ada source code from synchronous digraph real-time task models. Symbolic methods are applied to synthesize controllers for both discrete and real-time systems in [AMP95]. [KPP12] synthesize asynchronous reactive systems for multi-threaded environments. Hardware is automatically synthesized from specifications by solving two-player games in [Blo+07]. There is a lot of more work in this area. Yet, the publications mentioned above are orthogonal to our work, since we do not synthesize code or controllers, but adjust system variables.

The method applied in this work, probabilistic model checking for parametrized Markov models, is applied by Češka et al. in [Češ+17] to configure parameters of a stochastic biochemical network such that a given time-bounded continuous stochastic logic property is satisfied. Cubuktepe et al. apply probabilistic model checking to parametric variants of the models for a consensus protocol, a bounded retransmission protocol and NAND multiplexing [Cub+17]. Fine tuning of probabilistic parameters for self-stabilizing algorithms via PMC is considered by Aflaki et al. in [Afl+17].

There is more work in the field of configuration via both arbitrary formal methods and probabilistic model checking for parametrized Markov models. Giving a full list would go beyond the scope of this work. Focusing on fault tolerance, there is a lot of literature concerning fault-tolerance analysis. In the context of configuration there is a lot of work in the field of controller synthesis, but related work for configuration of system variables is rather sparse.

Fault-Tolerance Analysis via Formal Methods. Focusing on fault tolerance, there is a lot of literature about applying formal methods for resilience analysis. To name a few analysis approaches from the non-probabilistic setting, [YS05], present an approach to verify the correctness of a fault-tolerance mechanism for a set of interacting processes, and use formal methods to examine side effects of faults and fault tolerance. Also on the non-probabilistic and analytic side is the work of [EL16], where the fault tolerance of an IEEE 802.5 token ring LAN protocol is analyzed using model checking and fault injection. In [Baa+11], symbolic (non-probabilistic) model checking is used to quantify the robustness of sequential circuits. [FFR01] and [Sch+98] validate space-craft requirements via model checking.

Also in the probabilistic setting, formal methods have been widely used to analyze resilience properties. First to name here is the work of Kuvaiskii et al., who present a high-level CTMC model of HAFT in [Kuv+16], and use it to analyze the availability of HAFT. To name some more, the reliability of a NAND-multiplexer is evaluated in [Nor+04], and of FPGA implementations of an adder and a multiplier in [Hoq+14], both using the probabilistic model checker PRISM [KNP11]. [FYO14] verify and analyze a micro processor unit that is protected against noise by the use of a reset strategy. Ahmad et al. analyze the resilience of a satellite solar array in [AH16], Reliability Block Diagrams in [AHT16a], logistic service supply chains in [AHT16b], wireless sensor networks in [AHT15], and pipelines in [Ahm+14], using theorem proving and higher-order-logic formalizations of probability theory.

Fault-Tolerance Configuration via Formal Methods. Formal methods have been widely applied to synthesize fault-tolerant systems, e.g., in [GR09; DF09; Dum+07; Hsi00].

The feasability of formal methods for resiliency configuration of system variables has not been investigated very often so far. However, there are some approaches in the literature. In [HKM08], Han, Katoen, and Mereacre configure a reliable storage system, modeled as a continuous-time Markov chain, with respect to expected time between storage checks. The model suites the purpose of this paper, and demonstrates the usability of an approach that approximates optimal configurations for parametric continuous-time Markov chains with respect to time-bounded reachability properties. Yet, the model is rather high-level, and comprises rather high error rates. In [Wu+12], the authors propose a semi-automated technique to achieve resilience in a system consisting of several components. This approach requires the resilience of each component to be given, and identifies components where resilience increase has the most impact on the overall system resilience. Therefore, for each component, the resilience of this component is increased to 1, and the effect is analyzed. This is in contrast to our work, where we apply PMC on parametric models to obtain resilience characteristics for several configurations with one computation, and allow arbitrary resilience values as configuration goal.

Closely related to this work is the work of Gmeiner et al., who propose and analyze the combination of several existing techniques to apply model checking to configure fault-tolerant distributed algorithms in [Gme+14]. Yet, this work is on the non-probabilistic side.

Tackling the State-Explosion Problem. The state-explosion problem is a well-known problem in the field of probabilistic model checking, and thus, there are many existing techniques addressing this problem. To name a few prominent techniques, there is bisimulation [HM80; Seg95], a technique reducing the model size. Storm implements bisimulation, and thus we always apply this technique when invoking Storm. Counter abstraction [ET99] is a technique that can be applied to Markov models where some components have identical behavior. Counter abstraction provides a more compact representation for these components. Other techniques are predicate abstraction [WZH07], counter-example guided abstraction refinement [Cla+00], partial order reduction [Lip75; Maz87; Pel96; God96], and compositional methods like assume-guarantee reasoning [CLM89; GL91; AHJ01; Păs+08; Kwi+10]. [CJP18] used the model structure to tackle the state-explosion problem. They provide a component-based method to identify domain-specific modeling patterns in quality-of-service systems, extract sub-models from the modeling patterns, analyze the sub-models using parametric model checking, and combine the results to obtain overall characteristics. The method is semi-automated and needs domain-specific knowledge to be applied.

In Chapter 4, we tackle the state-explosion problem arising during the configuration of the redo-based fault-tolerance protocol with a new counter-based factorization technique. This technique is orthogonal to the previously mentioned approaches in the sense that these techniques can be applied on top of or alongside counter-based factorization.

1.8 Outline

Chapter 2 summarizes notations and preliminaries from the field of Markov models, property specification, and statistics. In Chapter 3, we present the exemplary configuration of inter-process communication structures. Chapter 4 introduces the factorization approach and its implementation in `fact`. The configuration of redo-based fault tolerance is exemplified in Chapter 5. Finally, in Chapter 6, we conclude this thesis and give a summary on open topics and further work.

1.9 Own Publications

The configuration of inter-process communication protocols for fault tolerance has been published at the workshop “Hot Topics on Operating Systems” (HotOS) in [Leu+17], and was extended in the journal “Foundations for Mastering Change” (FoMac) in [Her+18b]. First observations concerning accuracy and time issues of iterative methods are published at “Computer-Aided Verification” (CAV) in [Bai+17]. Configuration of redo-based fault tolerance has been investigated in [Her+18a] and published at the “European Performance Engineering Workshop” (EPEW). The counter-based factorization approach has not yet been published, yet a version tailored to the redo-based fault-tolerance protocol is included in [Her+18a]. Other work that does not appear in this theses, but is published in [Bai+14], presents the energy-utility analysis of a triple-modular redundancy technique.

Sections 3.1 to 3.5 have been originally published in [Her+18b]. Sections 5.2 and 5.3 are taken from [Her+18a]. Although other authors were involved in the publications [Her+18b] and [Her+18a], the sections used in this thesis originate from the author herself.

1.10 Resources

The model generation tool and the space probe model used in Chapter 3, the tool `fact`, and the models used for redo-based fault-tolerance configuration in Chapter 5 can be downloaded at `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/Thesis-Herrmann/`.

2 Preliminaries

In the further, we denote the set of natural numbers by $\mathbb{N}$, the set of rational numbers by $\mathbb{Q}$, and the set of non-negative rational numbers by $\mathbb{Q}^{\geq 0}$. We denote the power set of a set S by 2^S. We fix a non-empty set *Vars* of variables with finite, non-empty domain, and denote the domain of a variable $v \in \mathit{Vars}$, i.e., the set of values that can be assigned to v, by $\mathit{Dom}(v)$.

Let Ω be a non-empty set, and $\mathcal{F} \subseteq 2^\Omega$. A mapping $P\colon \mathcal{F} \to [0,1]$ with $\sum_{f\in\mathcal{F}} P(f) = 1$ is a probability distribution on $\mathcal{F}$. The set of all probability distributions on $\mathcal{F}$ is denoted by $\mathit{Distr}(\mathcal{F})$. Let $f \in \mathcal{F}$. The Dirac distribution (degenerate distribution) for f is the probability distribution $\mathit{dirac}_f \in \mathit{Distr}(\mathcal{F})$ such that $\mathit{dirac}_f(f) = 1$ and $\mathit{dirac}_f(f') = 0$ for all $f' \neq f$. Let $A, B \in \mathcal{F}$, and let $P \in \mathit{Distr}(\mathcal{F})$. The *conditional probability* of A given B is $P(A|B) = \frac{P(A\cap B)}{P(B)}$. Let $\mathcal{F}_1, \mathcal{F}_2 \subseteq 2^\Omega$ be non-empty, let $P_1 \in \mathit{Distr}(\mathcal{F}_1)$, and let $P_2 \in \mathit{Distr}(\mathcal{F}_2)$. The joint probability distribution $P_1 \times P_2 \in \mathit{Distr}(\mathcal{F}_1 \times \mathcal{F}_2)$ is given as: $P_1 \times P_2(f_1, f_2) = P_1(f_1) \cdot P_2(f_2)$ for $f_1 \in \mathcal{F}_1$ and $f_2 \in \mathcal{F}_2$. Let $A \in \mathcal{F}$. A series of sets $A_1, \ldots, A_k \in \mathcal{F}$ such that $A_i \cap A_j = \emptyset$ for all $1 \leq i < j \leq k$ and $\bigcup_{1\leq i\leq k} A_i = A$ is a *partitioning* of A.

Let AP be a finite set of atomic propositions. The set of propositional formulas over AP is denoted by $\mathrm{PL}(\mathsf{AP})$. Let f be a propositional formula. The syntax of a linear-time logical (LTL) formula is defined by the following BNF:

$$\varphi = \mathit{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathrm{U}\, \varphi,$$

where $a \in 2^{\mathsf{AP}}$. We use the usual abbreviations: $\mathit{false} = \neg\mathit{true}$, $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\Diamond\varphi = \mathit{true}\,\mathrm{U}\,\varphi$, and $\Box\varphi = \neg\Diamond\neg\varphi$. For the semantics of linear-time logic, we refer to, e.g., [BK08].

Probabilistic Control-Flow Graphs An evaluation of *Vars* is a mapping $\mathfrak{e}\colon \mathit{Vars} \to \bigcup_{x\in \mathit{Vars}} \mathit{Dom}(x)$ with $\mathfrak{e}(x) \in \mathit{Dom}(x)$ for all $x \in \mathit{Vars}$. The set of all evaluations over *Vars* is denoted by $\mathit{Eval}(\mathit{Vars})$. The projection of $\mathfrak{e}$ to a subset of variables $X \subseteq \mathit{Vars}$ is the evaluation $\mathfrak{e}{\downarrow_X} \in \mathit{Eval}(X)$ such that $\mathfrak{e}{\downarrow_X}(x) = \mathfrak{e}(x)$ for all $x \in X$. A *guard* over *Vars* is a propositional formula over $2^{\mathit{Eval}(\mathit{Vars})}$. An evaluation $\mathfrak{e}$ of *Vars* models a guard g on *Vars*, denoted by $\mathfrak{e} \models g$, if replacing every atomic proposition ξ in g with *true* if $\mathfrak{e} \in \xi$, and with *false* otherwise, yields a valid formula. An update on *Vars* is a mapping $\mathfrak{u}\colon \mathit{Eval}(\mathit{Vars}) \to \mathit{Eval}(\mathit{Vars})$. The set of all updates on *Vars* is denoted by $\mathit{Upd}(\mathit{Vars})$. We denote $\mathrm{id} \in \mathit{Upd}(\mathit{Vars})\colon \mathrm{id}(\mathfrak{e}) = \mathfrak{e}$ for all $\mathfrak{e} \in \mathit{Eval}(\mathit{Vars})$. Let $X, Y \subseteq \mathit{Vars}$ be disjoint, let $\mathfrak{u}_X \in \mathit{Upd}(X)$, and let $\mathfrak{u}_Y \in \mathit{Upd}(Y)$. The joint update $\mathfrak{u}_X \times \mathfrak{u}_Y$ on $X \cup Y$ is defined as: $\mathfrak{u}_X \times \mathfrak{u}_Y(\mathfrak{e})(x) = \mathfrak{u}_X(\mathfrak{e}{\downarrow_X})(x)$ for all $x \in X$ and $\mathfrak{u}_X \times \mathfrak{u}_Y(\mathfrak{e})(y) = \mathfrak{u}_Y(\mathfrak{e}{\downarrow_Y})(y)$ for all $y \in Y$. Note that for an update $\mathfrak{u} \in \mathit{Upd}(X \cup Y)$

there are unique updates $\mathfrak{u}_x \in \mathit{Upd}(X)$ and $\mathfrak{u}_y \in \mathit{Upd}(Y)$ such that $\mathfrak{u}_x \times \mathfrak{u}_y = \mathfrak{u}$. A probabilistic update on *Vars* is a mapping $\mathfrak{pu}$ in $\mathit{Distr}(\mathit{Upd}(\mathit{Vars}))$. The result of applying $\mathfrak{pu}$ to an evaluation $\mathfrak{e}$ of *Vars*, denoted by $\mathfrak{pu}(\mathfrak{e})$, is the set $\mathfrak{pu}(\mathfrak{e}) = \{\mathfrak{e}' \mid \mathfrak{e}' = \mathfrak{u}(\mathfrak{e}) \text{ for some } \mathfrak{u} \in \mathit{Upd}(\mathit{Vars}) \text{ with } \mathfrak{pu}(\mathfrak{u}) > 0\}$. The set of all probabilistic updates on *Vars* is denoted by $p\mathit{Upd}(\mathit{Vars})$. Let $X, Y \subseteq \mathit{Vars}$ be disjoint and let $\mathfrak{pu}_1 \in \mathit{Distr}(\mathit{Upd}(X)), \mathfrak{pu}_2 \in \mathit{Distr}(\mathit{Upd}(Y))$. The joint probabilistic update $\mathfrak{pu}_x \times \mathfrak{pu}_y$ on $X \cup Y$ is defined as: for all $\mathfrak{u} \in \mathit{Upd}(X \cup Y)$: $\mathfrak{pu}_x \times \mathfrak{pu}_y(\mathfrak{u}) = \mathfrak{pu}_x(\mathfrak{u}_x) \cdot \mathfrak{pu}_y(\mathfrak{u}_y)$, where $\mathfrak{u}_x \in \mathit{Upd}(X)$ and $\mathfrak{u}_y \in \mathit{Upd}(Y)$ are the unique updates such that $\mathfrak{u} = \mathfrak{u}_x \times \mathfrak{u}_y$.

Let $\mathit{LV} \subseteq \mathit{Vars}$. A *probabilistic program graph* (PPG) over *Vars* is a tuple $\mathcal{P} = (\mathit{LV}, \mathit{Act}, \hookrightarrow)$ where

- *Act* is a finite, non-empty set of *actions*, and
- $\hookrightarrow \subseteq \mathrm{PL}(\mathit{Vars}) \times \mathit{Act} \times p\mathit{Upd}(\mathit{LV})$ is a set of *transitions*.

A transition $(g, \alpha, \mathfrak{pu}) \in \hookrightarrow$ is denoted by $g \overset{\alpha}{\hookrightarrow} \mathfrak{pu}$. $\mathcal{P}$ is purely probabilistic, if for each evaluation $\mathfrak{e} \in \mathit{Eval}(\mathit{Vars})$ there is at most one transition $g \overset{\alpha}{\hookrightarrow} \mathfrak{pu}$ such that $\mathfrak{e} \models g$. A PPG that is not purely probabilistic is nondeterministic. A reward structure in $\mathcal{P}$ is a mapping $\mathit{rew}\colon \mathit{Eval}(\mathit{Vars}) \to \mathbb{Q}^{\geq 0}$.

When depicting a PPG, a designated variable is named *location*. This variable's evaluations are represented by named circles.

Let $\mathcal{P}_1 = (\mathit{LV}_1, \mathit{Act}_1, \hookrightarrow_1)$ and $\mathcal{P}_2 = (\mathit{LV}_2, \mathit{Act}_2, \hookrightarrow_2)$ be two PPGs over *Vars*. $\mathcal{P}_1$ and $\mathcal{P}_2$ are composable if $\mathit{LV}_1 \cap \mathit{LV}_2 = \emptyset$. The composition is the PPG $\mathcal{P}_1 || \mathcal{P}_2 = (\mathit{LV}, \mathit{Act}, \hookrightarrow)$ over *Vars* where $\mathit{LV} = \mathit{LV}_1 \cup \mathit{LV}_2$, $\mathit{Act} = \mathit{Act}_1 \cup \mathit{Act}_2$, and $\hookrightarrow \subseteq \mathrm{PL}(\mathit{Vars}) \times \mathit{Act} \times p\mathit{Upd}(\mathit{LV})$ is the smallest relation such that:

- for all $\alpha \in \mathit{Act}_1 \cap \mathit{Act}_2$: $$\frac{g_1 \overset{\alpha}{\hookrightarrow}_1 \mathfrak{pu}_1 \qquad g_2 \overset{\alpha}{\hookrightarrow}_2 \mathfrak{pu}_2}{g_1 \wedge g_2 \overset{\alpha}{\hookrightarrow} \mathfrak{pu}_1 \times \mathfrak{pu}_2}$$
- for all $\alpha \in \mathit{Act}_1 \setminus \mathit{Act}_2$: $$\frac{g_1 \overset{\alpha}{\hookrightarrow}_1 \mathfrak{pu}_1}{g_1 \overset{\alpha}{\hookrightarrow} \mathfrak{pu}_1 \times \mathit{dirac}_{\mathrm{id}}}$$
- for all $\alpha \in \mathit{Act}_2 \setminus \mathit{Act}_1$: $$\frac{g_2 \overset{\alpha}{\hookrightarrow}_2 \mathfrak{pu}_2}{g_2 \overset{\alpha}{\hookrightarrow} \mathit{dirac}_{\mathrm{id}} \times \mathfrak{pu}_2}$$

Note that the composition of purely probabilistic program graphs is not necessarily purely probabilistic.

Discrete-Time Markov Chains. Let AP be a set of atomic propositions. A discrete-time Markov chain (DTMC) over AP is a tuple $\mathcal{M} = (S, P, \mathsf{L})$ where S is a non-empty, finite set, $P\colon S \to \mathit{Distr}(S) \cup f_0$, where f_0 is the mapping assigning each state the value

0, and $\mathsf{L}\colon S \to 2^{\mathsf{AP}}$ is a labeling function. For all $s, t \in S$, we denote $P(s,t) = P(s)(t)$. For states $s, t \in S$, we say that there is a transition from s to t, if $P(s,t) > 0$. A state $s \in S$ with $P(s) = f_0$ is called absorbing. For details to Markov chains see, e.g., [BK08; Kul95].

A *path* in $\mathcal{M}$ that starts in $s_1 \in S$ is a finite or infinite sequence $\pi = s_1 s_2 \dots$ with $s_i \in S$. A path π is maximal, if it is infinite, or it is of the form $s_1 s_2 \dots s_n$ and s_n is absorbing. For a finite path $\pi = s_1 s_2 \dots s_n$ we write $\Pr(\pi) = \prod_{i=1}^{n-1} P(s_i, s_{i+1})$. Let $s \in S$. The set of all finite paths in $\mathcal{M}$ starting in s is denoted by $\mathit{FPaths}_{\mathcal{M},s}$, the set of infinite paths is $\mathit{IPaths}_{\mathcal{M},s}$, the set of maximal paths is $\mathit{MPaths}_{\mathcal{M},s}$, and the set of all paths is $\mathit{Paths}_{\mathcal{M},s}$. Let $\xi \subseteq S$. We denote $\Pi_s(\xi) = \{\pi = s_1 \dots s_n \in \mathit{FPaths}_{\mathcal{M},s} \mid s_n \in \xi, s_i \notin \xi \text{ for } 1 \leq i < n\}$.

Let $s \in S$ and φ be an LTL-property. The probability of φ in $\mathcal{M}$ with starting state s, denoted by $\Pr_{\mathcal{M},s}(\varphi)$, is derived using the standard definition of the induced probability distribution on the set of measurable sets of maximal paths. Let $G \subseteq S$. The LTL formula $\Diamond G$ is called a *probabilistic reachability property*. The steady-state probability of G is $\theta_{\mathcal{M},s}(G) = \lim_{n\to\infty} \frac{1}{n} \cdot \sum_{i=1}^{n} \theta_{i,\mathcal{M},s}(G)$ where $\theta_{k,\mathcal{M},s}(G) = \Pr(\{s_1 s_2 \dots s_k \in \mathit{FPaths}_{\mathcal{M},s} \mid s_k \in G\})$.

Rewards in DTMCs. A *reward function*, also called reward structure, is a mapping $\mathit{rew}\colon S \to \mathbb{Q}^{\geq 0}$. Let $s \in S$. The accumulated reward induced by rew is a random variable $\mathit{AccRew}\colon \mathit{Paths}_{\mathcal{M},s} \to \mathbb{Q}^{\geq 0}$ assigning each path in $\mathcal{M}$ the sum of its state rewards, i.e., $\mathit{AccRew}(s_1 \dots s_n) = \sum_{i=1}^{n-1} \mathit{rew}(s_i)$. Note that state rewards are collected when leaving a state. In PPGs, reward structures often are defined as mappings $\mathit{rew}\colon \mathit{Act} \to \mathbb{Q}^{\geq 0}$ or $\mathit{rew}\colon S \times \mathit{Act} \to \mathbb{Q}^{\geq 0}$. These notions are equivalently expressive. In this work we will also assign rewards to actions, and assume that the reader is familiar with techniques that allow for transforming action rewards into state rewards. Let $s \in S$ and $G \subseteq S$ be a set of states such that $\Pr_{\mathcal{M},s}(\Diamond G) = 1$. The *expected accumulated reward* until reaching G from s, denoted by $\mathbb{E}_{\mathcal{M},s}(\diamondplus G)$, is the expectation value of AccRew in $\mathcal{M}$ restricted to the set of paths ending in G, i.e., $\mathbb{E}_{\mathcal{M},s}(\diamondplus G) = \sum_{\pi \in \Pi_s(G)} \Pr(\pi) \cdot \mathit{AccRew}(\pi)$.

For $G \subseteq S$ and $\Pr_{\mathcal{M},s}(\Diamond G) > 0$ the *conditional expected accumulated reward* until reaching G from s under the condition of reaching G is:

$$\mathbb{E}_{\mathcal{M},s}(\diamondplus G \mid \Diamond G) = \sum_{\pi \in \Pi_s(G)} \frac{\Pr(\pi) \cdot \mathit{AccRew}(\pi)}{\Pr_{\mathcal{M},s}(\Diamond G).}$$

If $\Pr_{\mathcal{M},s}(\Diamond G) = 0$ we define $\mathbb{E}_{\mathcal{M},s}(\diamondplus G \mid \Diamond G) = 0$.

If the state s is clear from the context, e.g., if there is a designated initial state, we omit it's index in all of the presented notations.

Induced DTMC. Let $\mathcal{P} = (\mathit{LV}, \mathit{Act}, \hookrightarrow)$ be a purely probabilistic program graph over Vars such that $\mathit{LV} = \mathit{Vars}$. The semantics of $\mathcal{P}$ is the induced DTMC $\mathcal{M}_{\mathcal{P}} = (S, P, \mathsf{L})$ with $S = \mathit{Eval}(\mathit{Vars})$, $\mathsf{L}(\mathfrak{e}) = \mathfrak{e}$, and $P(\mathfrak{e}_1, \mathfrak{e}_2) = p$ if there is a transition

$g \overset{\alpha}{\hookrightarrow} \mathfrak{pu}$ of $\mathcal{P}$ such that $\mathfrak{e}_1 \models g$ and there is an update $\mathfrak{u} \in \mathit{Upd}(LV)$ such that $\mathfrak{u}(\mathfrak{e}_1) = \mathfrak{e}_2$ and $\mathfrak{pu}(\mathfrak{u}) = p$ and $P(\mathfrak{e}_1, \mathfrak{e}_2) = 0$ otherwise. We say that the DTMC $\mathcal{M}_{\mathcal{P}}$ *arises* from $\mathcal{P}$. Note that, since each state $s \in S$ is an evaluation of Vars, we can denote $s(x)$ to obtain the value of variable $x \in \mathit{Vars}$ in state s. A reward structure rew in $\mathcal{P}$ is also a reward structure in the induced DTMC.

The semantics of a non-deterministic program graph is a Markov decision process, i.e., a Markov chain enhanced with non-determinism. In this work, Markov decision processes will not be considered. For details to Markov decision process, we refer to, e.g., [BK08; Put94].

3 Configuration of Fault-Tolerant Inter-Process Communication

Communication of processes, when not handled properly, easily enables abuse and undesired access to other processes' memory. For this reason, *inter-process communication* (IPC) is typically implemented in the kernel of the underlying operating system, which provides and protects communication structures and handles access rights. Bit flips in this data can cause messages to be readable by the wrong process, data-loss, or the crash of single process or even the whole system. For this reason, fault tolerance needs to be ensured for communication on operating system level. The term *inter-process communication protocols* stands for a set of protocols coordinating communication of a finite number of processes. The protocols differ in the communication structures they provide and in the safety strategies and rights management implemented to protect these communication structures.

In this chapter we consider a set of n communicating processes that produce utility. The concrete type of utility depends on the processes' purpose. For example, several processes in a bank server and a process in bank branch terminal communicate with each other when a costumer draws some money. The utility is, e.g., the number of successful money transfers per hour.

Bit flips affect the communication structures and thus the utility, which is decreased due to error detection and correction overhead. The goal of this chapter is to demonstrate how to find the sweet spot, where the utility is maximized while still resilience is provided, for a concrete set of processes running in the L4/Fiasco.OC microkernel [16b]. The characteristics of the processes and the concrete communication structure can be specified easily via a domain specific language (DSL). We provide a tool that automatically generates a model from the DSL-description. The utility produced by the process network can then easily be specified in the model.

To demonstrate the configuration procedure, we consider a part of a space probe consisting of 4 processes. The example is inspired by the Maven Mars probe [Jak+15; 11a]. The modeled part is responsible for collecting and transferring data to earth. The space probe can not detect all errors that arise in its communication structures. Undetected errors remain in the system, and their effect is unpredictable. In a worst-case assumption no utility is produced by the system any longer, as soon as there is an undetected error in the system. For this reason, the space probe is restarted periodically to get rid of undetected errors. We model time in the space probe as discrete time, each modeled time step spans 100 micro seconds. In each time step a restart is triggered with a certain probability. This probability models heuristics that are used to determine the need of a restart, e.g., heuristics using the space weather or the overall state of the space probe. The probability defines the expected time between two restarts. As utility, we measure the availability of the space probe, i.e., the percentage of time where the space probe neither suffers from an undetected error, nor performs a restart. The configuration criterion is to maximize the utility of the system, by tuning the frequency of periodic restarts. Restarting to often will cause availability loss because of time that is needed to restart the system. Restarting to seldom cause availability loss due to undetected errors.

We first analyze standard PMC methods, show that iterative methods suffer from (well-known) accuracy and timing issues when applied to this model, and thus motivate the usage of non-iterative methods for configuration.

We then configure the space probe for a fixed error probability of 10^{-10}, i.e., within every 100 micro seconds there is a chance of 10^{-10} that a bit flip affects some communication structure. The restart probability per time step is used as a parameter during PMC. We also include the detection probability as a parameter, and we invoke PMC to find the rational function describing the availability of the space probe in dependence of the detection probability and restart probability. We analyze this rational function using Newton's method to find, for a fixed set of detection probabilities, the respective optimal restart probability.

We then analyze some characteristic of this optimally configured system, revealing that the optimal setting requires 99% of all restarts to be performed without having an error in the system.

Outline. In the first section of this chapter we introduce the inter-process communication protocol as used in the L4/Fiasco.OC microkernel [16b] and explain occurrence of errors and their effects. Then, in Section 3.2, we specify the model that arises from a concrete communication network, and we define the domain-specific language used to describe a set of communicating processes. In Section 3.3, we explain the space-probe example. Then we apply iterative and non-iterative PMC methods in Section 3.4, including parametric variants to compute selected properties of the space probe. We argue that iterative methods are not suitable to configure the space probe, even when not using system variables as parameters. Finally, in Section 3.5, we define the configuration criterion we want to address, configure the space probe part, and analyze the optimal configuration.

3.1 Inter-Process Communication in Fiasco.OC

In the IPC protocol used in the operating system Fiasco.OC communication is controlled by the kernel completely. Instead of a process A sending a message directly to a process B, the kernel provides IPC channels, to which A can send messages and from which B can receive messages. During message transfer, the sending process A is in the role of a "client" and the receiving process B is a "server". These roles can change with every new message transfer. For each IPC channel, the kernel maintains two buffers: Servers that are prepared to receive messages are stored in the

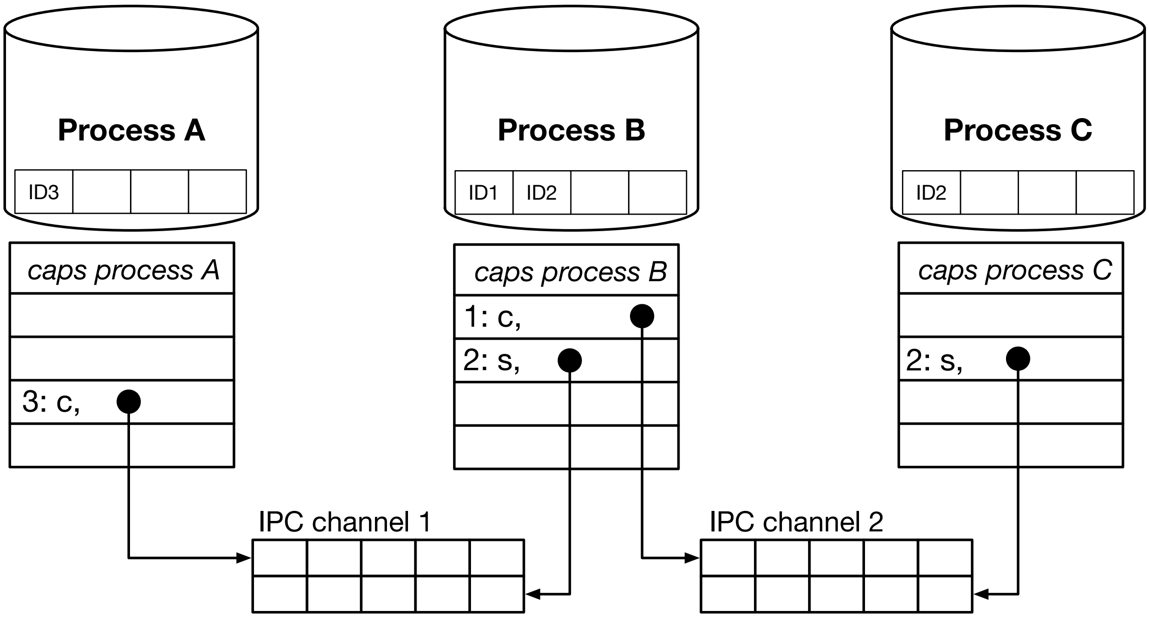

Figure 3.1: Capability-based inter-process communication of three processes in a pipe.

server buffer, and clients that want to send messages via this channel are collected in the client buffer. This way, A does not need to know which processes can receive messages from the channel. This indirect communication increases the anonymity of the communicating processes. For non-anonym communication, IPC channels between exactly two communicating processes are established.

The kernel grants access to some IPC channel by handing over capabilities. A capability is a pointer to the IPC channel's memory, enhanced with some additional information, e.g., the type of access right. A process can only send (receive) messages via some IPC channel if it has a capability providing client (server) access to this channel.

In Fiasco.OC, security is improved by not handing over capabilities directly, but only IDs for capabilities. This way, a malevolent process can not easily corrupt the capability. When some process intends to use a capability, it performs a system call, providing the capability ID. The kernel maintains for each process a table which maps capability IDs to concrete capabilities. If a process tries to use a capability ID which is not listed in the table, the kernel throws an error.

In this chapter, we consider two types of processes: processes that process inquiries (e.g., printers or operating system services) and processes that make inquiries (e.g., user processes). Former ones start in the role of a server, might become clients during inquiry processing, and become a server again after inquiry processing. Latter one are processes that always are in the role of a client (they only send messages and thus have no capability providing server access). We name this processes “client-only processes”. Furthermore, we assume communication networks to be cycle-free.

Example 3.1. *In Figure 3.1, three processes, A, B, and C, and IPC channels 1 and 2 form a pipe. Process B has two capabilities, listed in the capability table below B. The first capability (which is known to B as the capability with ID 1), grants client (sending) access to IPC channel 2. Client access is depicted by the small "c" in the respective entry of the capability table. The second capability grants server (receiving) access to channel 1, denoted by a small "s". Process A, a client-only process, can send messages to channel 1 and Process C can receive messages from channel 2.*

Message transfer. There are three participants in each message transfer: the client (sending process), the server (receiving process), and the kernel which checks access rights and copies the data from the client's memory to the server's memory.

The workflow of a client is simple: It performs a system call `send(IDx)`, providing the ID of the channel it wants to use for message transfer. After this, the client is blocked until it receives an answer to the sent message.

A process becomes a server when performing a system call `receive(IDx)`, informing the kernel that it is ready to receive messages via the channel represented by ID x. Then, the server is blocked until it receives a message. After message receipt, it can either send an answer to the respective client directly, or it might send a message to another process beforehand. In the latter case, the former server becomes a client and behaves as described above. After being unblocked, the process becomes a server again and now might send an answer to the original message. For this, the server performs a system call `answer_receive(IDx)`, signalizing the kernel that the answer can be copied into the client's memory, and that the server is ready to receive a new message via the channel with the respective ID x.

The kernel is responsible for checking access rights and transferring data during communication. When receiving a system call (`send(IDx)`/`receive(IDx)`) it blocks the calling process, checks whether the ID refers to an entry in the process' capability table, which IPC channel it refers to, and whether the respective capability grants client/server access to this channel. If not, the kernel throws an error and unblocks the process. Otherwise, the kernel adds the calling process to the IPC channels client/server buffer and waits until a server/client is ready to receive/send a message via this channel. As soon as there is an entry in both the channel's client buffer and the channel's server buffer, the kernel copies the message from the client's memory into the server's memory, and unblocks the server. Eventually, the kernel receives a system call `answer_receive(IDx)` from some server. It blocks the server, copies the server's answer to the client's memory, and checks whether the ID grants server access to some IPC channel. Note that answering does not involve any IPC channel, thus the server does not need to provide a capability for answering, only for the next message receive.

Example 3.2. *Figure 3.2 depicts an example message flow through the pipe in Example 3.1. Process A sends a message to Process B, which, before answering the message, sends a message to process C. Process A first performs a system call*

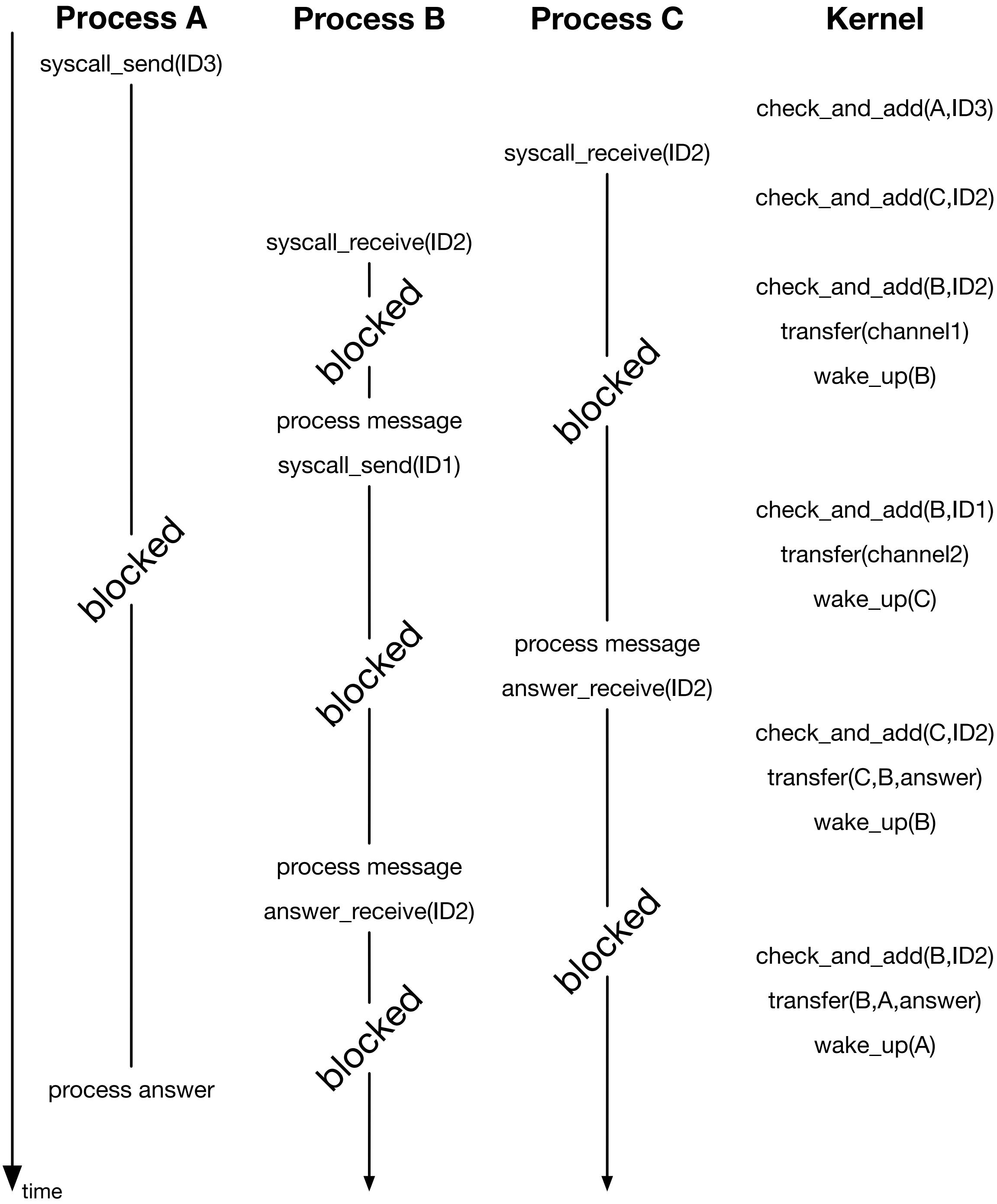

Figure 3.2: Message flow and process/kernel activities during IPC. A sends a message to B, which, for processing this message sends a message to C. C answers B's message, and finally B answers A's message.

`send(ID3)`. The kernel blocks process A and checks, whether ID3 refers to an entry in A's capability table, which IPC channel (channel 1) it refers to, and whether the respective capability grants client access to this channel. Additionally, the kernel adds process A to the IPC channels client buffer in this step. A's message can not yet be transferred, since no server is ready to receive the message.

Process C then performs a system call `receive(ID2)`. This causes C to be blocked and, since ID2 of process C refers to channel 2, process C to be added to channel 2's server buffer. Still, no message can be transferred, since for channel 1 only a client is prepared to receive a message and for channel 2 only a server is prepared.

In the next step, B performs system call `receive(ID2)`, and thus is blocked and added to channel 1's server buffer. Now, the kernel can copy the message from process A's memory to process B's memory and wake up process B. Process B processes the received message, but, before answering, needs to send a message via channel 2. Thus, B becomes a client, performing system call `send(ID1)`. B is blocked, and since process C is in channel 2's server buffer, the message from B can be transferred to C. After this, C is unblocked, prepares an answer for B, and performs system call `answer_receive(ID2)`. With this system call, the answer is copied from C's memory to B's memory, and C is added to channel 2's server memory, to receive a new message (after checking that ID2 is a valid ID, as before). B is unblocked, proceeds with processing A's message and prepares an answer. In this phase, B could also send another message via channel 2, but in our example, it performs system call `answer_receive(ID2)`, and thus is blocked and added to channel 1's server buffer. The kernel copies the answer from server B's memory to client A's memory and wakes up A, which then can process B's answer. In the further, A can again send a message via channel 1, and so on.

Errors. We focus on randomly occurring bit flips that affect communication structures, i.e., the IPC IDs maintained by the processes, the capability tables maintained by the kernel, or the memory of the IPC channel. Most of these bit flips, if affecting the IPC protocol at all, are easily detectable. If, for example, a capability in the table maintained by the kernel is affected, there is a high chance of the erroneous capability pointing to invalid memory. When a process performs a system call, the kernel checks whether the included IDs are valid and whether the respective capability points to valid memory. If a bit flip causes these data to be (detectably) corrupted, the kernel will throw an error. Other errors in other data structures used for message transfer (e.g., channel buffers) can also lead to error messages, and correction mechanisms can be applied. Nevertheless, there is a chance that a bit flip changes memory in a non-detectable way, i.e., to cause a silent data corruption. In this case the behavior of the system is completely indeterminate.

Example 3.3. *Continuing Example 3.1, it could happen that the capability pointer of B's capability 1 is corrupted by a bit flip. This bit flip might be detectable (e.g., if the corrupted pointer points to some invalid memory), or it might be a silent data* *corruption (e.g., if it points to some other channel). In the latter case, a message from B is misdirected to some other process than process C, which unintentionally has access to data from B.*

A safe way to react to detected errors is to restart the system. After a restart the system is completely fresh, and all errors are gone. To handle silent data corruptions, periodic restarts that are independent from the error states are performed.

3.2 Modeling Fault-Tolerant Inter-Process Communication

The model consists of components for processes, the kernel, and IPC channels, as well as a component modeling error occurrence and silent data corruptions, and a component modeling restarts.

All components are program graphs that interact via synchronous actions. In the pictures we mark actions that are triggered by the depicted component with an explanation mark (!) and actions in components that react are marked with a question mark (?). Recall that transitions having the same name are performed synchronously. Each action is marked with the name of all components that synchronize on this action. This way, we ensure that no unintended synchronization occurs. Nevertheless, in the rest of this section we will omit these additional information when they are clear from the context. Thus, instead of writing, e.g., “SEND_process_A_to_process_B_via_channel_i”, we simply write “SEND”.

Processes. Each process alternates working phases and message phases. The model abstracts from concrete work but only models time that passes by. After working, the process performs a system call, which activates the kernel component and causes the process to be blocked until being woken up by the kernel. When awakened, the process starts another working phase (cf. Figure 3.3).

The models of a client-only process and a process that can act as a server differ only in the initial location. A client-only process’ initial location is “working” (cf. Figure 3.3). In this location, it works for expectably `working_time`, time units. This is modeled using a geometric distribution, i.e., in each step there is a chance 1/`working_time` to finish work and to reach location “work done”. As a last part of its working step, there is a probabilistic choice in the capability that shall be used for message transfer. The probability of choosing ID x is `p_cap_id`. The process performs a system call, informing the kernel that it intends to send a message with capability x. The probability `p_message` is only relevant for processes that can be servers and thus set to 1 for all client-only processes. The kernel is activated and the process is blocked (location “blocked as client”) until the receiving server sends an answer. After this, the kernel unblocks the process with action “WAKE UP”. The process then enters its next working phase and proceeds as before. Note that a client-only process will never reach location “blocked as server”, since `p_message` $= 1$.

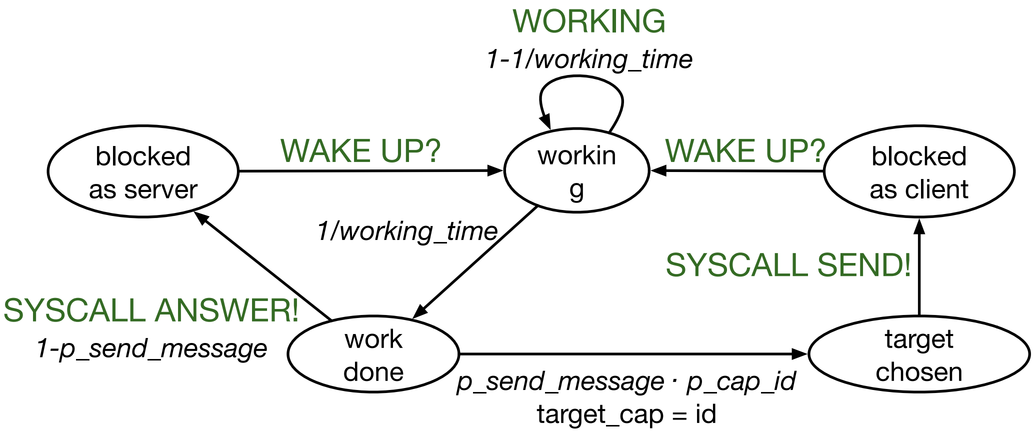

Figure 3.3: The PPG of a process. If the process is a client-only process then `p_send_message` = 1.

All processes that have a server capability are initially servers waiting for messages, i.e, they start in location "blocked as server". They are unblocked after the kernel copied a message to their internal memory and performs action "WAKE UP". Analogously to the client-only processes, the server works for some time and then performs a system call. For a server, there is a probabilistic choice in whether it sends an answer directly (with probability 1 − `p_message`) or whether it first calls another server (with probability `p_message`). In the latter case, it becomes a client and behaves the same way client-only processes do, i.e., it is blocked until it receives an answer. If it sends an answer directly, it is also blocked after the system call, but as a server, i.e., it is ready to receive new messages and will be woken up as soon as there is a new message for this process.

The action names of the system calls in Figure 3.3 are annotated with the IPC channel IDs in use, to enable correct synchronization (see paragraph "IPC channels" below).

Kernel. We split up the model of the kernel in several components, one for each process' kernel activity. For each process, the respective kernel component, depicted in Figure 3.4, is initially inactive, and is invoked when the process performs a system call, i.e., "SYSCALL SEND" or "SYSCALL ANSWER". Sending a message or an answer takes some time which is again modeled via a geometric distribution. If during message transfer an error is detected (not depicted in Figure 3.4, cf. paragraph "Errors"), the kernel component stops its activities and the restart component is invoked. With this, all other components but the restart component also stop their activities. (See paragraph "Restart" below.)

If no error is detected, the kernel successfully transfers the message or answer. If a message was transferred (i.e., a client performed the system call), then this kernel component waits for another kernel component to transfer the answer for this message. After this answer is transferred, the kernel wakes up the corresponding process.

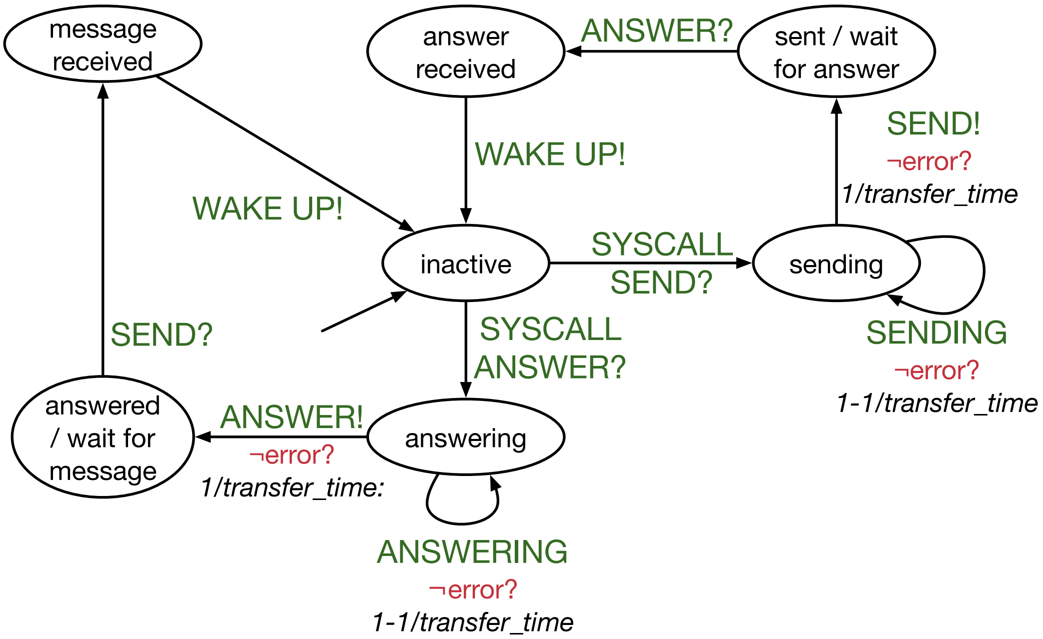

Figure 3.4: The Markov model of the kernel's activity for some process.

In the other case, if the process performed "SYSCALL ANSWER", the kernel transfers the answer and reaches location "answered/wait for message". The kernel component waits until another kernel component transfers a new message to the process' memory and then wakes up the process.

IPC channels. IPC channels do not have internal behavior. These data structures consists only of two FIFO buffers, one for enqueuing clients that aim to send messages via this channel, but need to wait because no server is ready, and one buffer to enqueue servers that are ready to receive messages. In a concrete communication network, we know how many clients and servers have capabilities for this channel, thus we set the maximal queue sizes to be these numbers. In reality, the buffer sizes are chosen large enough, so that buffer overflows can not happen. The IPC components synchronize on "SYSCALL SEND", "SEND", and "ANSWER" actions. When "SYSCALL SEND" is performed by some client, the ID of the IPC channel to be used is already known (cf. Figure 3.3), so the respective channel synchronizes on this action. The client is added to the channel's client buffer. When the kernel performs action "ANSWER" for some server, the server finished proceeding the message and is ready to receive the next one. Thus, the server is added to the respective IPC channel's server buffer.

Since the buffers are FIFO buffers, an IPC channel component blocks all "SEND" actions via this channel that involves clients or servers that are not first in line. Thus, when action "SEND" is performed, a message is transferred from the first client in the buffer to the first server in the buffer. Both the client and the server are removed from the buffers in this step, enabling other clients and servers to transfer messages via this channel.

Errors. The error component synchronizes on actions where time elapses, i.e., on all "WORKING", "SENDING" and "ANSWERING" actions. In each of these transitions there is a chance `p_error` of a bit flip causing a (detectable) error or a silent data corruption (sdc). The probability of an error being detectable, given an error occurred, is `p_detect`. In our model, for each IPC channel, there is a Boolean variable modeling whether there is a detectable error in some data referring to this channel. We assume that in each time step, and thus during each synchronization on "WORKING", "SENDING" or "ANSWERING", only one channel can be affected by an error. Thus, when synchronizing on one of the upper mentioned actions, for each channel there is a chance of $\texttt{p\_error} \cdot \texttt{p\_detect/num\_IPC\_channels}$ that this channel is affected by an error, where `num_IPC_channels` is the total number of modeled IPC channels.

A detectable error is only detected when the channel is used. Thus, a detectable error remains in the system until the respective channel is used for message transfer. Because of this, although only one single error can occur in a time step, it is possible to have multiple channels to be in an erroneous state at the same time.

When synchronizing on a time-consuming action, there is a chance $\texttt{p\_error} \cdot (1 - \texttt{p\_detect})$ that an error occurs and this error manifests as a silent data corruption. This is modeled via a (channel-independent) Boolean variable sdc, which is initially false. Since effects of silent data corruption are unpredictable, we made the worst-case assumption that no useful system functionality is obtained after an sdc. We did not model any system functionality after a silent data corruption.

Restarts. When an error is detected by some kernel component, the whole system except for the restart component stops. The restart component performs a restart, which takes expectably `restart_time` time units (cf. Figure 3.5). When finishing the restart, it performs an action "RESTART", where all other component synchronize. To correctly model process inactivity during restarts, all components but the restart component are modified in the following way (not depicted in the figures): If the restart component is in its "restart = true" location, all actions of other components are blocked, and a single action "RESTART" is enabled. This action resets all locations and variables to the initial value.

To recover from an sdc, we include periodic, probabilistic restarts in our model. In every transition modeling time elapsing there is a chance of the system to restart, including time that passes by after an sdc. Again, after a restart the system's functionality is completely regained.

To model these periodic restarts, the component synchronizes on actions modeling time passing by, and in each transition there is a chance `p_restart` to trigger a periodic restart. If this happens, analogously to the restart in case of a detected error, the whole system but this component stops, and the restart is performed. Note that the error component does not synchronize on action "RESTARTING". Thus, in our model it is not possible that an error manifests during a restart. Analogously, it is not possible that a restart is interrupted by another periodic restart.

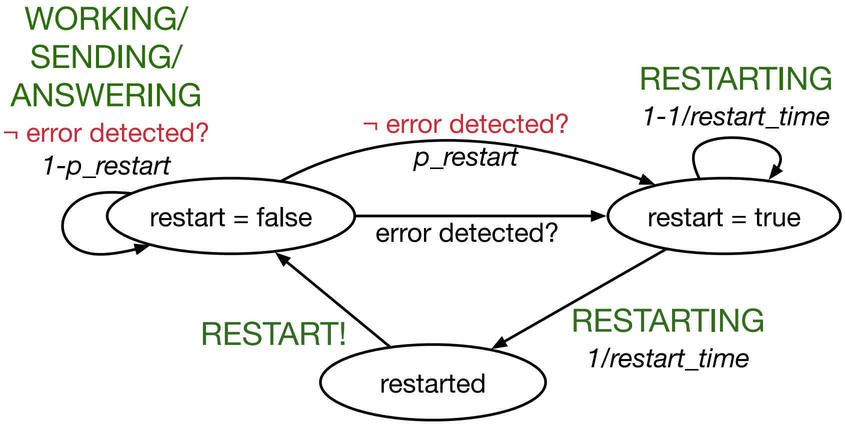

Figure 3.5: The Markov model of a the restart component.

A DSL for IPC In the domain-specific language, there is one block for each process (for an example, cf. Figure 3.6). A process block starts with the Identifier "Process" and the name of the process. Each process has a set of capabilities, each consisting of a name, an ID, and the access type it grants ("client" or "server"). Each capability needs to be listed in a separate line. Furthermore, for each process the following information needs to be provided, each in a separate line:

- The `working_time` (in time units), preceded by the identifier "Runtime".
- The probability of a server to send a new message instead of sending an answer directly, given by the identifier "prob_send_message", followed by the process' `p_message`.
- If `p_message` > 0, the probabilities `p_cap_id` of sending a message via IPC channel with name m. The structure for this information is: "prob_target: m, `p_cap_id`". This construct also defines for client-only processes the probability distribution over IPC channels that can be accessed.

The probabilities `p_error`, `p_restart`, and `p_detect` are provided as command-line arguments when starting the generation tool. Probabilities will be handled as free parameters when providing names instead of rational values.

The DSL enables short and easy description of communication networks. The model generation tool retrieves from the DSL description the number of processes, the number of channels, and the communication structure, and automatically generates a model for this concrete setting. This facilitates the modeling process. Furthermore, since the IPC functionality does not need to be modeled for each concrete communication structure, the usage of the DSL and the automatic model generation reduces the risk of human flaws during the modeling process.

```
Process:control
  Cap:name=con,id=3,access=client
  Runtime:25
  prob_send_message:1
  prob_target:con,1
Process:analysis
  Cap:name=t,id=2,access=client
  Cap:name=m,id=1,access=client
  Cap:name=con,id=3,access=server
  Runtime:5
  prob_send_message:0.98
  prob_target:t,0.02
  prob_target:m,0.98
Process: measure
  Cap:name=m,id=1,access=server
  Runtime:1
  prob_send_message:0
Process:transfer
  Cap:name=t,id=2,access=server
  Runtime:50
  prob_send_message:0
```

Figure 3.6: Representations of the space probe in the domain-specific language.

3.3 The Space-Probe Example

Figure 3.6 shows the DSL-description for the space probe, which we will use as an example throughout the rest of this chapter. One time unit in the model, i.e., one transition modeling time, represents 100 microseconds. The space probe part is capable of measuring data in space and transferring this data to Earth. It consists of four processes (cf. Figure 3.7).

A process `Control` periodically invokes this part of the space probe by alternating between working for expectably 2.5 milliseconds and sending messages to process `Analysis`. `Analysis` coordinates a process `Measure`, which collects data, and a process `Transfer`, which transfers data to earth. `Control` is a client-only process,

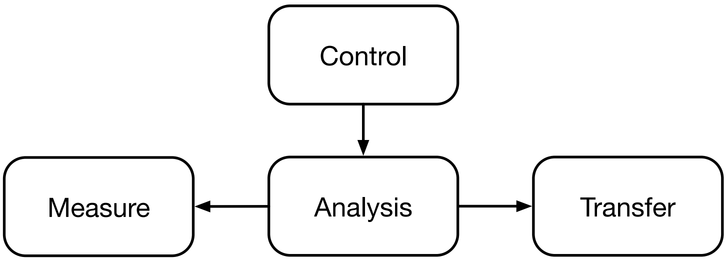

Figure 3.7: The communication structure of the space-probe example.

i.e., it can, when not blocked, produce messages without receiving another message before. All other processes are initially blocked, waiting for a message.

When receiving a message from `Control`, `Analysis` is unblocked. It performs some work for expectably 500 microseconds. After this, a message is sent to either `Measure` or `Transfer` with probability 0.98 (`p_message`). With probability 0.02 it directly sends an answer to `Control`. In the former case, a message is sent to `Measure` with probability 0.98. `Measure` is unblocked, collects data in space for expectably 100 microseconds and then sends an answer to `Analysis`. `Analysis` is unblocked when receiving the answer from `Measure`, and, after the working phase, might either send an answer to `Control` with probability 0.02, send another message to `Measure` with probability $0.98 \cdot 0.98$, or with probability $0.98 \cdot 0.02$ it chooses to send a message to `Transfer`. If so, `Transfer` transfers the collected data to Earth, which takes expectably 5 milliseconds. It then sends an answer to `Analysis`, which again has the choice of sending an answer to `Control` or re-invoking one of the other processes. When sending an answer to `Control`, `Control` is unblocked by the kernel, which starts a new cycle.

The probability of an error occurring within 100 microseconds is 10^{-10}. The probability of initiating a restart within 100 microseconds and the probability of detecting an error are left as free parameters. A restart lasts expectably 1 second.

The Composition is Purely Probabilistic. In the composition of all component PPGs, an action is available, if the action is available in all components that implement this action. All components but the kernel components are, when standing alone, purely probabilistic. Only in the kernel component there is non-determinism in the available actions in the “inactive” location. When composing all components, this non-determinism is no longer present, since in the respective process components there is no non-determinism in this actions. Thus, in the composed model, non-determinism arises only from different interleavings of the components.

The composition of the space probe components is purely probabilistic, since the order of execution of the components is purely probabilistic: There is only one client-only process (`Control`), and there is only one server for each IPC channel. Thus, when starting the protocol, only `Control` is unblocked, its kernel process is inactive, and all other processes are blocked and their respective kernel components wait for a message that can be transferred. `Control` will eventually send a message to `Analysis`, which unblocks `Analysis`. At this time, `Control`, `Measure`, and `Transfer` are blocked and their respective kernel components wait for messages. `Analysis` probabilistically chooses a target IPC channel, which again causes only one process to be unblocked in the next message transfer, and all other processes are blocked while their kernel components wait for message transfer. Similar arguments reveal that at any time there is only one unblocked process and in each state there is no non-determinism in the choice which process shall be unblocked next (this choice is made before and was probabilistic). Thus, the induced semantics of the composed model is a DTMC.

3.4 Time Consumption and Accuracy of Iterative PMC Methods

Parametric versions of PMC that accept parametric Markov models as input use non-iterative methods to solve the linear equation system arising during PMC. Despite the advantages of these parametric versions, non-iterative methods often consume more time than iterative methods. In this section we argue the usage of iterative methods is not advisable for models comprising very small transition probabilities, as it is the case when modeling bit-flip probabilities.

A parameter-free Markov chain can be analyzed using numerical solvers, e.g., value iteration [Bel57]. It is well-known that value iteration, when equipped with a termination ε, can not guarantee that the computed result is within an ε-neighborhood of the theoretical result [HM18]. During the work on [Leu+17], we discovered that the space probe model is affected by accuracy issues when computing accumulated rewards and reachability probabilities. In this section, we give a more detailed insight into this issue and show that, for the space probe example, inaccuracy increases when some transition probabilities tend to 0 or 1. We apply value iteration to a series of error probabilities. Decreasing the error probability causes the probability of not having an error in some working step to approach 1 and the probability of having an error to shrink. Thus, with lower error probability, transition probabilities are more close to 0 and 1.

We zoom into several scenarios and reveal differences in the usability of the inaccurate results provided by value iteration. Furthermore we apply interval iteration [HM18] and compare time consumption and convergence.

Setting. The space-probe model's semantics is a Markov chain $\mathcal{M}$ and consists of 877 states. The bisimulation quotient has 145 states. We used the model checker Storm [Deh+17]. Storm uses a non-iterative method by default. When forcing Storm to use value iteration, the default termination ε is 10^{-6}, and the starting value for iteration is 0.5. Computation was performed on a 2.5GHz Intel Core with 16GB RAM single-threaded. We use `p_detect` = 0.4, `p_restart` = $7.7456 \cdot 10^{-8}$ as non-parametric setting[1], and error probabilities ranging from 10^{-3} to 10^{-15}. We consider both the absolute termination criterion $x_i - x_{i+1} < \varepsilon$ and the relative termination criterion $(x_i - x_{i+1})/x_{i+1} < \varepsilon$, for succeeding interim results x_i and x_{i+1}.

Inaccuracy. We start with analyzing the effect of value-iteration inaccuracy for an expected accumulated reward property. We use a reward structure assigning action ANSWER_from_process_`Transfer`_to_process_`Analysis`_via_channel_2 the value 1, i.e., we count the number of transfers to earth. Table 3.8 shows the time consumption and the inaccuracy of the results when computing the expected accumulated reward $\mathbb{E}_{\mathcal{M}}(\lozenge \mathrm{sdc})$, i.e., the expected number of transfers to Earth until the first silent data corruption arises. The table summarizes the computed results, the computation time, and the number of

---

[1] In Section 3.5, we will see that this is an optimal setting guaranteeing an availability of 99.9%.

iterations for both the relative and the absolute termination criterion. For comparison, the correct values computed with (accuracy-loss-free) LU-elimination and the time consumption of this method are included.

We see that, when using the relative termination criterion with the standard termination bound $\varepsilon = 10^{-6}$, for all error probabilities below 10^{-6} there is a significant deviation between the exact result and the result computed with value iteration. For example, for `p_error` $= 10^{-6}$ and $\varepsilon = 10^{-6}$, value iteration computed an accumulated reward of 1805.8, while the correct value is 3459.4. Inaccuracy increases when decreasing the error probability. For `p_error` $= 10^{-15}$, value iteration computed results that are not inaccurate in some digits but in factors above 1000, even when $\varepsilon = 10^{-9}$. Note that, since `p_restart` $= 7.7456 \cdot 10^{-8}$, for all error probabilities there are transition probabilities close to 0 and 1 in the model. Thus, accuracy problems with value iteration do not always arise when transition probabilities are close to 0 and 1. Instead, this depends on the model and the analyzed property.

Using an absolute termination criterion instead of a relative one produces much better results. For example, for `p_error` $= 10^{-9}$ and $\varepsilon = 10^{-6}$, the computed result is 3459221, when using absolute termination, which is within a relative $\varepsilon = 10^{-3}$ neighborhood of the correct result. The result computed with relative termination criterion is 5972. For expected accumulated rewards reaching high values, this increase of accuracy is reasonable. A high nominator causes the relative distance to be very small and thus boosts too early termination. Value iteration with absolute termination criterion performs more iterations and thus produces better results. This comes, of course, at the price of long computation times (see paragraph below). Results couldn't be computed for error probabilities 10^{-9} and lower, since computation time exceeded 10 hours.

We now switch to the analysis of accuracy for a probabilistic reachability property. Table 3.9 lists results, time consumption, and the number of iterations until termination for $\Pr_{\mathcal{M}}(\text{healthy} \, \mathrm{U} \, \text{restart})$, i.e., we compute the probability of the system being error-free and having no sdc until the first restart is performed. For this property in both modes, with relative and absolute termination, results are inaccurate when choosing `p_error` $\leq 10^{-6}$. Again, accuracy decreases with decreasing `p_error`. For `p_error` $= 10^{-9}$ and below, Storm performs only 6 iterations before meeting the standard termination criterion. More than 10^7 iterations are needed for the first digit to be correct. This causes the computed result to be off by 0.5 when choosing the standard termination ε and `p_error` $\leq 10^{-9}$. But even for $\varepsilon = 10^{-9}$, the result is accurate only in the first digit. Thus, for this reachability property value iteration is not suitable.

[HM18] proposed interval iteration as a solution for the value-iteration-caused inaccuracy. Interval iteration is guaranteed to be accurate within the proposed termination ε. This is ensured by choosing two starting values, one upper bound and one lower bound of the result. Iteration is performed twice, approaching the result from below and above. When the (relative or absolute) distance of the interim results is below ε, the iterative algorithm stops. This way it is guaranteed that enough iterations are performed to reach the desired accuracy. Nevertheless, this method reveals that performing enough iterations costs an unreasonable amount of time (see next paragraph).

`p_error`	ε	**result** (relative)	**time** (relative)	**iterations** (relative)	**result** (absolute)	**time** (absolute)	**iterations** (absolute)
10^{-3}	10^{-6}	2.9168	0.05 s	2810	2.9183	0.01 s	8858
10^{-3}	10^{-7}	2.9191	0.05 s	10552	2.9192	0.02 s	11602
10^{-3}	10^{-8}	2.9193	0.05 s	13296	2.9193	0.02 s	14346
10^{-3}	10^{-9}	2.9194	0.05 s	16040	2.9194	0.02 s	17090
10^{-3}	exact	2.9194	0.03 s	–	–	–	–
10^{-6}	10^{-6}	1805.8	0.8 s	$7.9 \cdot 10^5$	3458.4	7.5 s	$8.6 \cdot 10^6$
10^{-6}	10^{-7}	3127.1	2.1 s	$2.5 \cdot 10^6$	3459.3	10.3 s	$1.1 \cdot 10^7$
10^{-6}	10^{-8}	3423.0	4.2 s	$4.9 \cdot 10^6$	3459.4	12.6 s	$1.4 \cdot 10^7$
10^{-6}	10^{-9}	3455.8	6.5 s	$7.3 \cdot 10^6$	3459.4	14.8 s	$1.6 \cdot 10^7$
10^{-6}	exact	3459.4	0.04 s	–	–	–	–
10^{-9}	10^{-6}	5972	1.7 s	$1.8 \cdot 10^6$	3459221	2 h	$8.6 \cdot 10^9$
10^{-9}	10^{-7}	41743	11.2 s	$1.3 \cdot 10^7$	?	>10 h	?
10^{-9}	10^{-8}	296721	82.9 s	$9.6 \cdot 10^7$	?	>10 h	?
10^{-9}	10^{-9}	1674696	627.8 s	$7.1 \cdot 10^8$	?	>10 h	?
10^{-9}	exact	3460286	0.05 s	–	–	–	–
10^{-12}	10^{-6}	5992	1.7 s	$1.8 \cdot 10^6$	?	>10 h	?
10^{-12}	10^{-7}	42418	11.5 s	$1.3 \cdot 10^7$	?	>10 h	?
10^{-12}	10^{-8}	324509	87.25 s	$1.0 \cdot 10^8$	?	>10 h	?
10^{-12}	10^{-9}	3242019	897.5 s	$1.0 \cdot 10^9$	?	>10 h	?
10^{-12}	exact	$3.46 \cdot 10^9$	0.06 s	–	–	–	–
10^{-15}	10^{-6}	5992	1.7 s	$1.8 \cdot 10^6$	?	>10 h	?
10^{-15}	10^{-7}	42422	11.7 s	$1.3 \cdot 10^7$	?	>10 h	?
10^{-15}	10^{-8}	324538	87.8 s	$1.0 \cdot 10^8$	?	>10 h	?
10^{-15}	10^{-9}	3245020	895.3 s	$1.0 \cdot 10^9$	?	>10 h	?
10^{-15}	exact	$3.46 \cdot 10^{12}$	0.07 s	–	–	–	–

Table 3.8: Time consumption and results of probabilistic model checking in Storm when using Gauss-Seidel value iteration to compute an expected accumulated reward.

`p_error`	ε	**result** (relative)	**time** (relative)	**iterations** (relative)	**result** (absolute)	**time** (absolute)	**iterations** (absolute)
10^{-3}	10^{-6}	$4.3 \cdot 10^{-5}$	0.05 s	8072	$4.9 \cdot 10^{-4}$	0.06 s	3308
10^{-3}	10^{-7}	$4.3 \cdot 10^{-5}$	0.05 s	9164	$8.8 \cdot 10^{-5}$	0.06 s	4396
10^{-3}	10^{-8}	$4.3 \cdot 10^{-5}$	0.06 s	10256	$4.7 \cdot 10^{-5}$	0.01 s	5488
10^{-3}	10^{-9}	$4.3 \cdot 10^{-5}$	0.06 s	11347	$4.3 \cdot 10^{-5}$	0.01 s	6580
10^{-3}	exact	$4.3 \cdot 10^{-5}$	0.1 s	–	–	–	–
10^{-6}	10^{-6}	0.07489	0.4 s	$1.1 \cdot 10^{6}$	0.49037	0.06 s	9509
10^{-6}	10^{-7}	0.04320	0.8 s	$2.4 \cdot 10^{6}$	0.08617	0.3 s	$1.0 \cdot 10^{6}$
10^{-6}	10^{-8}	0.04144	1.0 s	$3.5 \cdot 10^{6}$	0.04575	0.7 s	$2.1 \cdot 10^{6}$
10^{-6}	10^{-9}	0.04127	1.3 s	$4.5 \cdot 10^{6}$	0.04170	0.8 s	$3.1 \cdot 10^{6}$
10^{-6}	exact	0.04126	0.1 s	–	–	–	–
10^{-9}	10^{-6}	0.500001	0.05 s	6	0.50000	0.05 s	6
10^{-9}	10^{-7}	0.500106	0.05 s	2341	0.50002	0.1 s	322
10^{-9}	10^{-8}	0.883311	5.1 s	$1.7 \cdot 10^{7}$	0.87090	5.1 s	$1.6 \cdot 10^{7}$
10^{-9}	10^{-9}	0.967001	11.6 s	$4.1 \cdot 10^{7}$	0.96665	11.2 s	$4.0 \cdot 10^{7}$
10^{-9}	exact	0.977289	0.1 s	–	–	–	–
10^{-12}	10^{-6}	0.500001	0.05 s	6	0.50000	0.05 s	6
10^{-12}	10^{-7}	0.500126	0.05 s	2724	0.50002	0.06 s	330
10^{-12}	10^{-8}	0.901804	5.1 s	$1.8 \cdot 10^{7}$	0.89111	5.3 s	$1.7 \cdot 10^{7}$
10^{-12}	10^{-9}	0.989208	12.8 s	$4.2 \cdot 10^{7}$	0.98909	11.8 s	$4.2 \cdot 10^{7}$
10^{-12}	exact	0.9999768	0.1 s	–	–	–	–
10^{-15}	10^{-6}	0.500001	0.05 s	6	0.50000	0.05 s	6
10^{-15}	10^{-7}	0.500126	0.06 s	2724	0.50002	0.06 s	330
10^{-15}	10^{-8}	0.901823	5.0 s	$1.8 \cdot 10^{7}$	0.89113	5.0 s	$1.7 \cdot 10^{7}$
10^{-15}	10^{-9}	0.989231	13.3 s	$4.2 \cdot 10^{7}$	0.98911	11.9 s	$4.2 \cdot 10^{7}$
10^{-15}	exact	0.99999998	0.01 s	–	–	–	–

Table 3.9: Time consumption and results of probabilistic model checking in Storm when using Gauss-Seidel value iteration to compute the probability of a reachability property.

`p_error`	ε	**result** (relative)	**time** (relative)	**iterations** (relative)	**result** (absolute)	**time** (absolute)	**iterations** (absolute)
10^{-3}	10^{-3}	2.920077	0.3 s	241021	2.919859	0.6 s	250249
10^{-3}	10^{-4}	2.919428	0.4 s	300181	2.919406	0.7 s	309407
10^{-3}	10^{-5}	2.919363	0.5 s	359295	2.919361	0.7 s	368521
10^{-3}	10^{-6}	2.919357	0.5 s	418409	2.919356	0.9 s	427635
10^{-3}	exact	2.919355	0.03 s	–	–	–	–
10^{-6}	10^{-3}	3460.807	19.7 s	$1.9 \cdot 10^7$	3459.444	79.3 s	$3.4 \cdot 10^7$
10^{-6}	10^{-4}	3459.580	29.3 s	$2.4 \cdot 10^7$	3459.444	89.3 s	$3.9 \cdot 10^7$
10^{-6}	10^{-5}	3459.457	36.1 s	$2.8 \cdot 10^7$	3459.443	72.0 s	$4.4 \cdot 10^7$
10^{-6}	10^{-6}	3459.445	33.7 s	$3.2 \cdot 10^7$	3459.443	109.9 s	$4.9 \cdot 10^7$
10^{-6}	exact	3459.443	0.04 s	–	–	–	–

Table 3.10: Time consumption and results of probabilistic model checking in Storm when using Gauss-Seidel interval iteration to compute the probability of an expected accumulated reward.

Time consumption and number of iterations. First note that there is a strong correlation in time consumption and number of iterations. Each iteration can be performed very quickly (about 10^6 iterations per second for the expected reward, about $3.5 \cdot 10^6$ iterations per second for the reachability probability). For the presented properties and the space-probe model, value iteration convergence is very slow, i.e., the difference between two succeeding intermediate results is very small. Especially for higher ε, this causes the termination criterion of value iteration to be fulfilled too soon and value iteration to stop too early. The amount of iterations to be performed, such that the result is in the ε-neighborhood of the result, can be determined using interval iteration. Table 3.10 shows results for the expected accumulated reward property. Note that ε ranges from 10^{-3} to 10^{-6}. The table shows that, when decreasing ε, the number of iterations needed until termination increases steadily, but relatively slowly. In comparison, when decreasing `p_error`, the number of iterations increases significantly, e.g., from $2.4 \cdot 10^5$ for `p_error` $= 10^{-3}$ to $1.9 \cdot 10^7$ for `p_error` $= 10^{-6}$ for relative termination and $\varepsilon = 10^{-3}$. Computation for `p_error` $\leq 10^{-9}$ did not terminate for any of the chosen ε within 10 hours.

Summary. The tables show clearly that iterative methods are not suitable to compute characteristics of the space-probe model due to the small error probabilities.

The exact LU-elimination outperforms almost all settings. In fact, this non-iterative method performs very well on our model, which is due to the small size of the linear equation system (145 lines) obtained from the model and the properties. This also enables and encourages the use of parametric versions of PMC, since parametric variants use non-iterative methods during model checking.

Unfortunately, the non-suitableness of iterative methods is not generalizable to other models. Inaccuracy of value iteration and time consumption of value iteration do depend on the small probability `p_error`, but do not depend on the small probability `p_restart`. Thus, these methods might scale for other models with small probabilities, e.g., other models considering fault tolerance. Yet, one should apply iterative methods with care to these models and always consider non-iterative methods as an alternative.

3.5 Configuration of Resilient Communicating Processes

The space probe presented in Section 3.3 has two system variables. The probability per time step of performing a periodic restart, `p_restart` shall be configured with respect to the probability of detecting an error, `p_detect`. In this section we apply parametric model checking techniques with two free parameters to configure the space probe for an error probability of $\texttt{p\_error} = 10^{-10}$, and we report on interesting system properties revealed during the configuration process.

Although the detection probability can be arbitrary close to one in theory [Sha48a], in reality typically there are restrictions imposed by memory- or hardware characteristics, and the existence of fault-tolerance techniques. We exemplarily choose the detection rate to be a value in $\{0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99\}$, representing, e.g., several fault-tolerance techniques that can be chosen to protect the space probe against bit flips. We do not impose any restrictions on `p_restart`. As the main configuration criterion we maximize the availability of the space probe. The availability of a system is the degree to which this system is operating correctly. It is typically given as a percentage, e.g., Amazon guarantees for its hosting service “Amazon EC2” an availability of 99.99% per month [18a], Google’s service level agreement for its cloud storage service lists an availability of 99.95% [16a]. The space probe is available when no restart takes place and no silent data corruption causes the system to be unpredictable. We compute $\theta(\mathrm{Av}) = \sum_{s\in\mathrm{Av}} \theta(s)$ with Av being the set of all states where the space probe is available and $\theta(s)$ being the steady-state probability of being in state s. We aim to configure the space probe’s system variable `p_restart` such that the availability is maximized.

We use aforementioned `p_restart` and `p_detect` as free parameters in probabilistic model checking and obtain a rational function describing the availability of the space probe. Furthermore, we report on interesting system properties revealed during the configuration process. For other properties than availability, we apply PMC with only one free parameter. PMC with two free probabilistic parameters is very time-consuming (see below). We use the computer algebra system sympy [Meu+17] to point-wise evaluate the rational functions and to obtain plots. We also use sympy to determine the optimum of the availability curve for the fixed detection rates using Newton’s method.

Availability maximization. The availability of the space probe is depicted in Figure 3.11. Figure 3.12 is a magnification to the optima of the curves. We see that, e.g., a detection probability of at least 0.4 is necessary to guarantee an availability of 0.999.

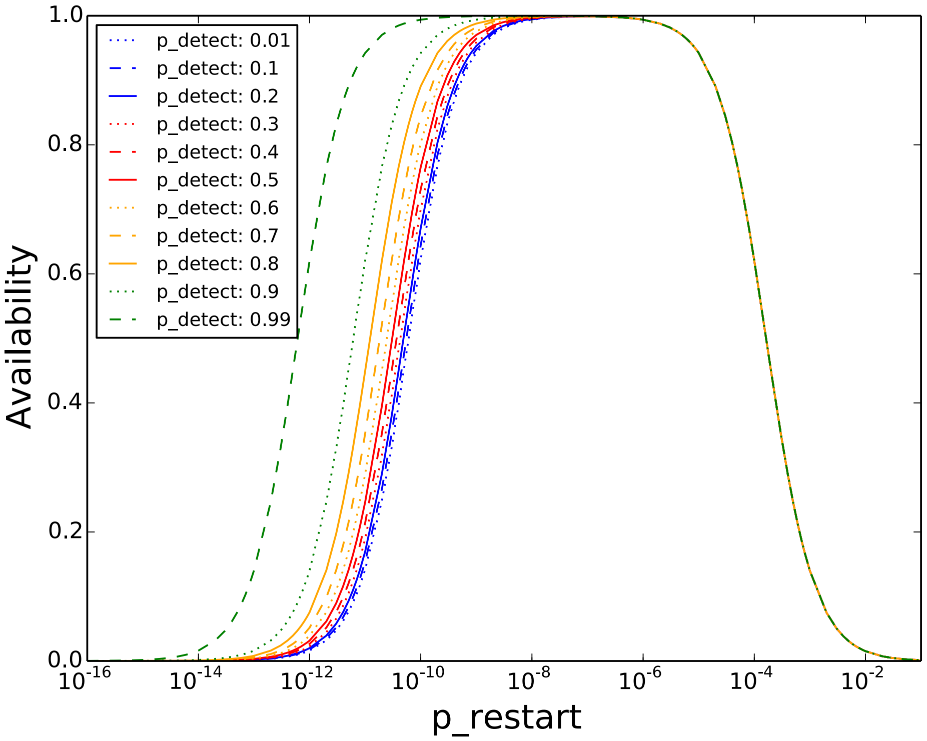

Figure 3.11: The availability of the space probe.

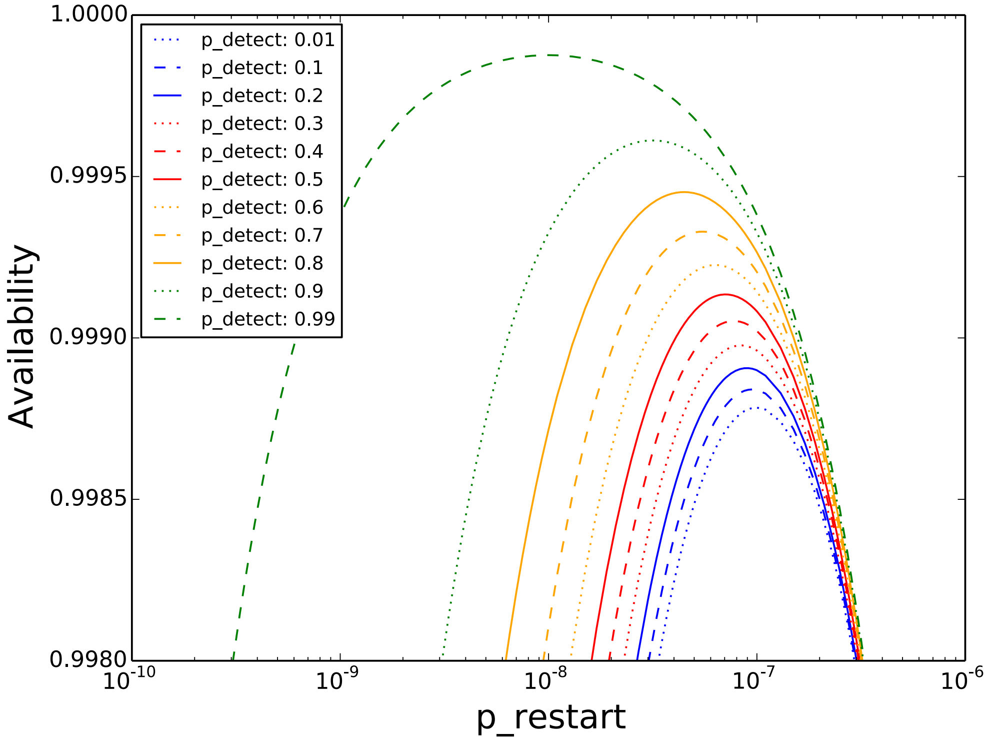

Figure 3.12: A magnification of the space probe's availability showing high availabilities.

Now we benefit from one of the most significant advantages of parametric methods. Instead of, for a given detection probability, vaguely reading the optimal restart probability from the picture, we analyze the rational function obtained from parametric model checking. We import the rational function into sympy, appoint the chosen detection rate to `p_detect`, and differentiate the resulting rational function. Applying Newton's method to find the zero of this function yields the results listed in Table 3.13. We can read the optimal setting for each of the chosen detection probabilities from the table. For example, the optimal `p_restart` for `p_detect` $= 0.4$ with respect to the previously mentioned configuration criterion is `p_restart` $= 7.7456 \cdot 10^{-8}$, i.e., restarts should be performed about every 21 hours and 30 minutes to obtain an availability of 99.9%.

`p_detect`	optimal `p_restart`	availability
0.01	$9.9494 \cdot 10^{-8}$	0.9987841
0.1	$9.4864 \cdot 10^{-8}$	0.9988405
0.2	$8.9438 \cdot 10^{-8}$	0.9989066
0.3	$8.3662 \cdot 10^{-8}$	0.9989769
0.4	$7.7456 \cdot 10^{-8}$	0.9990526
0.5	$7.0707 \cdot 10^{-8}$	0.9991348
0.6	$6.3242 \cdot 10^{-8}$	0.9992258
0.7	$5.4770 \cdot 10^{-8}$	0.9993292
0.8	$4.4719 \cdot 10^{-8}$	0.9994518
0.9	$3.1621 \cdot 10^{-8}$	0.9996117
0.95	$2.2356 \cdot 10^{-8}$	0.9997249
0.99	$9.9995 \cdot 10^{-9}$	0.9998759

Table 3.13: Optimal values for `p_restart` for varying detection probabilities, and the corresponding system availability.

Space-probe analysis. First note that for a fixed error probability `p_error` $\neq 0$ the system's availability is bounded to a value strictly less than one (see also Figure 3.14). This is because errors will affect the system, either in a detectable or a non detectable way. Both types of errors cause the system to be unavailable for some time: After an error is detected, the system is restarted and thus is unavailable for the time that is needed for a restart. Silent data corruptions cause unavailability for the time until the next periodic restart *and* the time of this restart. Since increasing `p_detect` decreases the steady-state probability of having a silent data corruption, (θ(sdc), cf. Figure 3.15), the availability can be increased by increasing the detection probability. The maximal (theoretically achievable) availability is gained when every error can be detected (`p_detect` $= 1$, no sdc occurs) and thus no periodic restart needs to be performed (`p_restart` $= 0$). In this setting the availability bound is 0.9999981656.

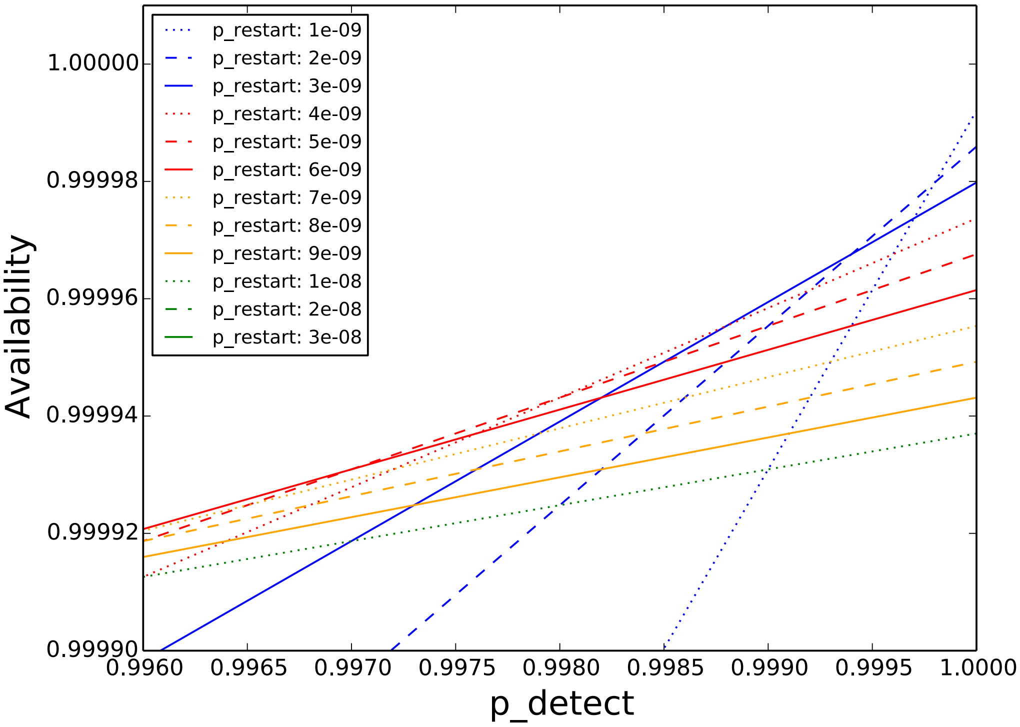

Figure 3.14: The availability of the space probe in dependence of the detection probability. For each restart probability, the availability runs to a value less than one, when increasing `p_detect`.

Figure 3.11 shows that with decreasing the probability `p_restart` the availability first rises and then drops down to zero again. This can be explained as follows: When choosing very high restart rates, too much time is spent for restarts, time that can not be used for system functionality. Performing periodic restarts too infrequently causes silent data corruption phases to be very long. The sweet spot depends on the detection probability, since this probability influences the probability of silent data corruptions (cf. Figure 3.15). This dependency also causes the crossings in Figure 3.14. On the left-hand side of a crossing, i.e., for a lower detection probability, it is more beneficial to choose a higher restart probability, since the low detection rate comes with a higher frequency of silent data corruptions, and periodic restarts should also be more frequent. Increasing the detection probability decreases sdc frequency, so on the right-hand side of a crossing it is more beneficial to choose a lower restart probability.

The values of the curves in Figure 3.11 on the right-hand side of the maxima are very close, the distance shrinking with higher restart probabilities. The concrete values only marginally depend on the detection rates. Figure 3.16 shows that this effect also arises when

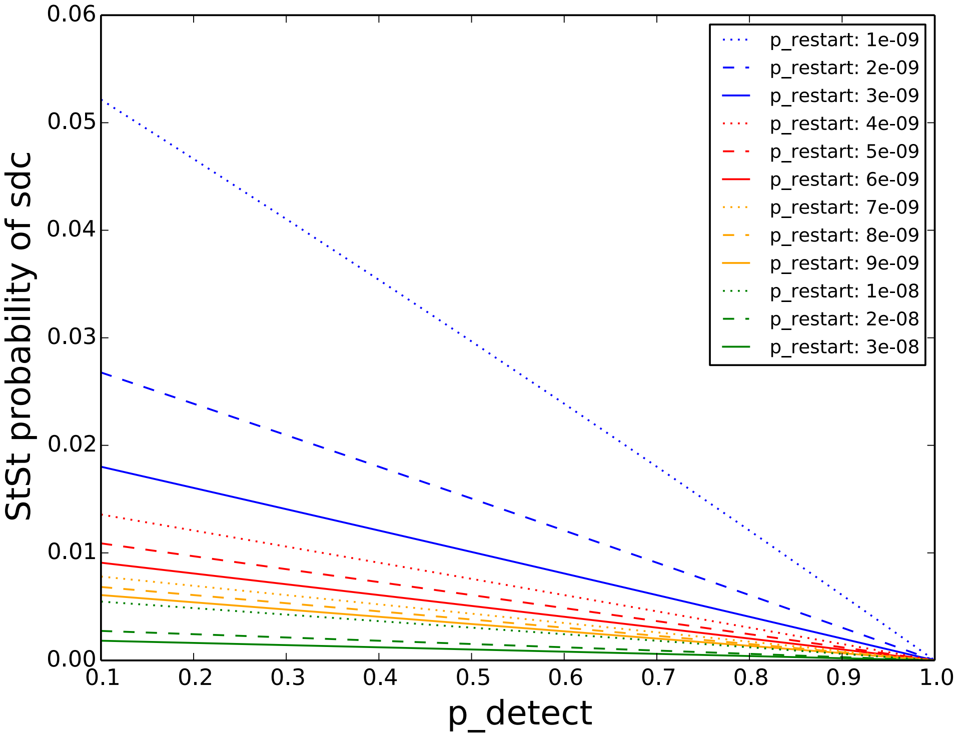

Figure 3.15: The steady state probability of a silent data corruption in dependence of the detection probability.

varying error probabilities. This means that availability is lost only marginally because of errors and their effect, but mainly because of the system being ill-configured: When restarts are performed too often, the system is unavailable mainly because of unnecessary restarts.

Figure 3.17 shows the (steady-state) probability of a restart being performed although the system neither comprises a detectable error nor a silent data corruption. Formally, we compute the steady-state probability $\theta(\text{restart} \wedge \text{healthy})/\theta(\text{restart})$, where restart is the set of all states where a restart is performed and healthy is the set of all states where neither a silent data corruption nor a detectable error is in the system. For high `p_restart`, this probability is close to one, and persistently decreases with decreasing restart rates. Interestingly, even for the optimal configurations about 98-99 out of 100 restarts are performed although the system is healthy (see Table 3.18). This means, that a configuration with maximal availability requires the acquiescence of many unnecessary restarts. A high detection probability causes this value to shrink, but the effect is only marginal.

Computation times. The space-probe model is a Markov chain and consists of 877 states. The bisimulation quotient has 145 states. We used the model checker Storm [Deh+17] to perform experiments on a machine with a 2.5 GHz Intel Core i7 CPU with 16 GB RAM single-threaded.

Table 3.19 lists computation times needed to compute the rational functions from above, the size of this rational functions in both the number of symbols, and the storage size in megabyte (MB). The table lists measurements when having one and when having two free parameters. When using only `p_restart` as free parameter we computed a set of rational

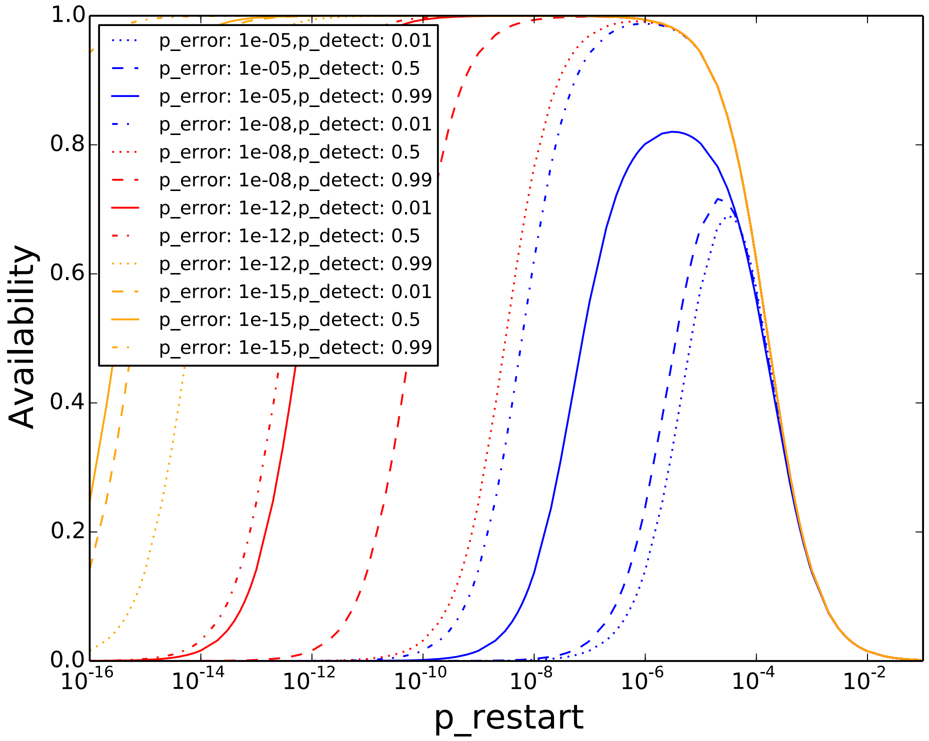

Figure 3.16: The availability of the space probe in dependence of the restart probability, for varying error probabilities and error detection probabilities.

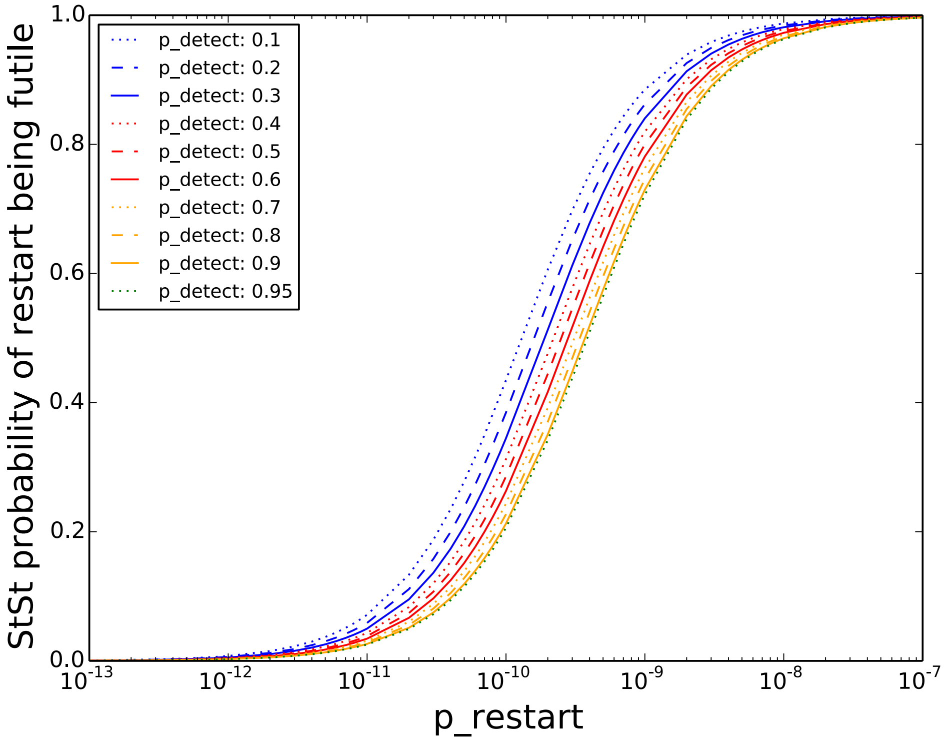

Figure 3.17: The ratio of unnecessary restarts and all restarts.

`p_detect`	**optimal `p_restart`**	**unnecessary restarts (percentage)**
0.1	$9.4864 \cdot 10^{-8}$	0.99863
0.2	$8.9438 \cdot 10^{-8}$	0.99821
0.3	$8.3662 \cdot 10^{-8}$	0.99773
0.4	$7.7456 \cdot 10^{-8}$	0.99717
0.5	$7.0707 \cdot 10^{-8}$	0.99648
0.6	$6.3242 \cdot 10^{-8}$	0.99559
0.7	$5.4770 \cdot 10^{-8}$	0.99437
0.8	$4.4719 \cdot 10^{-8}$	0.99245
0.9	$3.1621 \cdot 10^{-8}$	0.98843
0.95	$2.2356 \cdot 10^{-8}$	0.98307

Table 3.18: For optimal `p_detect`–`p_restart` combinations, the probability of a restart being performed although the system is neither affected by an error nor by a silent data corruption.

Property	**free param.**	**time**	**num symbols**	**size(MB)**
θ(sdc)	1	≈20 s	≈ 70000	0.070
θ(sdc)	2	8090 s	$2.8 \cdot 10^6$	2.8
θ(Av)	1	≈15 s	≈ 72000	0.072
θ(Av)	2	22 996 s	$2.9 \cdot 10^6$	2.9
θ(restart)	1	≈1400 s	≈ 66000	0.06
θ(restart)	2	>30 h	?	?
θ(restart $\wedge$ healthy)	1	≈1300 s	70000	0.07
θ(restart $\wedge$ healthy)	2	>30 h	?	?

Table 3.19: Time consumption and length of rational functions.

functions with varying `p_detect`. The table lists representative values for these sets of functions.

For the first two listed properties (availability and steady-state probability of having an sdc), computation with one free parameter was really fast (about 20 seconds). For the properties focusing on restart states, computation took 23 minutes. Performing PMC with two free parameters took about 6.5 hours for the first two properties listed in Table 3.19, but did not complete within 30 hours for the last two.

The table also shows that, when having two parameters, the size of the results is large (about 3MB). Evaluation of these functions is time-consuming, e.g., simply computing a concrete value of θ(Av) by appointing concrete values for `p_restart` and `p_detect` took about 4 seconds. Obtaining a rational function for a concrete `p_detect` from θ(Av) also took about 5 seconds. Differentiating the obtained function with sympy took less than a second. Applying Newton's method to obtain the zero of the differentiate took about 2 seconds.

It is well known, that a) time complexity of PMC on parametric models is exponential in the number of parameters and b) the actual time consumption depends not only on the model structure, but also on the property. This is reflected in our model, since parametrized versions of PMC have an acceptable runtime when considering only one free parameter. When considering two parameters, the runtime crucially depends on the concrete property. For our main configuration criterion, the availability of the space probe, we applied PMC to a model with free parameters `p_detect` and `p_restart`, which caused a long runtime, but we had the benefit of plotting curves fast.

4 Counter-based Factorization

In the previous chapter we configured a relatively small model for communicating processes using parametric variants of probabilistic model checking. In the next chapter, Chapter 5, we configure a parametric model for redo-based fault tolerance protecting a long-running application (for model details, see Section 5.2). This model is by far larger: depending on the chosen setting, the model has up to 10^{12} states. The model is actually so large, that it can not be built in Storm or Prism, not to mention performing PMC on this model. The complexity of the model arises from the application's run-time. The runtime is modeled by a counter, i.e., a variable whose value is never decreasing. The counter in the probabilistic program graph modeling the redo-based fault-tolerance protocol has a very large domain and thus causes the size of the induced DTMC to be huge.

In this chapter, we introduce counter-based factorization, a mathematical framework that allows to make use of the structure that arises from the counter. The model is split into small sub-models, PMC is applied to these sub-models to obtain rational functions describing local characteristics, and these results are combined to obtain a rational function describing the global configuration criterion that is used for optimization. We give the mathematical background of this framework, and present `fact`, a tool that implements factorization such that it can be applied to configure redo-based fault tolerance.

Factorization. The factorization framework presented in this chapter is applied to the induced DTMC of an arbitrary PPG having a counter variable (i.e., it is not restricted to the redo-based fault-tolerance model). We focus on the computation of probabilistic reachability properties and expected accumulated rewards. The idea of counter-based factorization is to flag states that are reached by transitions that increase the counter and to split the model along these states into factors. Since the counter is a variable that is always increased, these factors form a chain. For a probabilistic reachability property, for each factor the rational functions describing probabilities of reaching one of the flagged states in the next factor from this factor are computed. These local results are stored in a matrix. Since the factors form a chain, the matrices can be multiplied to obtain the rational function for the global probabilistic reachability property. When computing an expected accumulated reward, additionally the conditional expected accumulated rewards until reaching a flagged state in the next factor need to be computed. Then, the expected accumulated reward can be derived by computing for all factors the product of probability matrices of previous factors multiplied with the conditional expected accumulated reward matrix of this factor and summing up these results.

For the redo-based fault-tolerance model, the model structure arising from the counter is very specific. First, the counter increases, when updated, always by 1. We say that counters having this property are *simple*. Second, the counter is *observing*, i.e., the counter value does not influence the models structure, transition probabilities or reward assignments.

Hence, the matrices for local probabilistic characteristics of factors are all identical, and so are the matrices for local accumulated reward characteristics.

Implementation. The tool `fact` implements counter-based factorization for a simple, observing counter. It takes as input a probabilistic program graph in the common Prism-language [11b], together with a probabilistic reachability property or an expected accumulated reward property. `fact` analyzes the input PPG to find counters that are simple and observing and heuristically chooses one to apply factorization. A new variable f, which is used to identify flagged states in the induced DTMC, is inserted in the PPG. `fact` transforms the PPG such that it represents the factors of the induced DTMC, invokes the probabilistic model checker Storm on these factors, and sets up the local matrices. Finally `fact` makes use of the computer algebra system sympy to retrieve the global rational function from the local matrices.

Outline. In the first section of this chapter, Section 4.1, we define the basic, counter-independent, concept of factorization in arbitrary DTMCs. We assume two sub-models (factors) to be given. Section 4.2 introduces counters in PPGs, and shows how they can be used to obtain factors of the induced DTMC. The counter-based factorization approach arises from applying the general factorization approach for two factors from Section 4.1 recursively. In Section 4.3 we adopt the factorization approach to counters that are simple, and counters that are simple and observing. Finally, in Section 4.4 we present the tool `fact`.

4.1 Base Schema of Factorization

In this section, we explain the basic principle of factorization. We fix a DTMC $\mathcal{M} = (S, P, \mathsf{L})$, an initial state $s_{init} \in S$, a reward structure *rew*, and a set of states $\xi \subseteq S$. The state set S is assumed to contain only states that are reachable from s_{init}. Recall that $\Pi_{s_{init}}(\xi) = \{\pi = s_1 \dots s_n \in \mathit{FPaths}_{\mathcal{M},s_{init}} \mid s_n \in \xi, s_i \notin \xi \text{ for } 1 \leq i < n\}$. We assume that states $s \in \xi$ are absorbing, i.e., $\sum_{t \in S} P(s,t) = 0$. We present factorization for computing $\Pr_{\mathcal{M},s_{init}}(\lozenge\xi)$ and $\mathbb{E}_{\mathcal{M},s_{init}}(\oplus\,\xi)$. Recall that the index s is omitted when $s = s_{init}$.

The basic factorization approach operates on two sets S_1 and S_2 partitioning the state space S. This partitioning induces for each path of $\mathcal{M}$ two sub-paths: The first sub-path is a prefix of the original path that consists only of states in S_1 and a single state in S_2, and the second sub-path consists of the rest of the original path. With factorization, reachability characteristics and (conditional) expected accumulated rewards are computed on the sets of sub-paths, and the results are combined to obtain the values or rational functions for $\Pr_{\mathcal{M},s_{init}}(\lozenge\xi)$ and $\mathbb{E}_{\mathcal{M},s_{init}}(\oplus\,\xi)$.

Given a partitioning S_1, S_2 of the set of states S, we define the set of boundary states. Boundary states are states in S_2 that are reachable within one step from S_1. These states are the states where paths of $\mathcal{M}$ are split into sub-paths.

Definition 4.1 (Boundary States). *Let S_1, S_2 be disjoint sets such that $S_1 \cup S_2 = S$ and $s_{init} \in S_1$. The set of* boundary states *is*

$$S_b = \{t \in S_2 \mid \text{there is a state } s \in S_1 \text{ such that } P(s,t) > 0\}$$

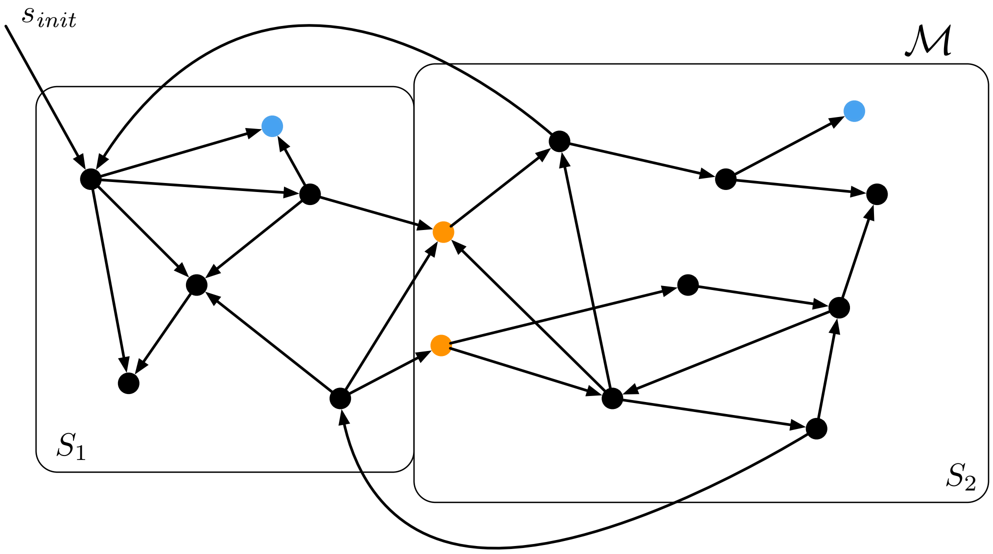

Figure 4.1: A DTMC $\mathcal{M}$ with initial state s_{init}, and a set of states ξ (blue circles). The boundary states arising from the partitioning S_1, S_2 are marked orange.

Figure 4.1 depicts the set of boundary states for a partitioning S_1, S_2. With factorization, the computation of reachability properties and expected accumulated rewards is split along the boundary states into sub-computations.

Lemma 4.2 (Factorization). *Let S_1, S_2 be disjoint sets such that $S_1 \cup S_2 = S$ and $s_{init} \notin S_2$, and let S_b be the set of boundary states. Let $\mathcal{M}'$ be the DTMC arising from $\mathcal{M}$ by making all states in S_2 absorbing. Then:*

$$\Pr_{\mathcal{M}}(\Diamond\xi) = \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \Pr_{\mathcal{M}, s_b}(\Diamond\xi). \tag{FP}$$

Let $\Pr_{\mathcal{M}}(\Diamond\xi) = 1$. *Then:*

$$\begin{aligned} \mathbb{E}_{\mathcal{M}}(\diamondplus\,\xi) = {} & \mathbb{E}_{\mathcal{M}'}\left(\diamondplus\left(\xi \wedge \neg S_b\right) \mid \Diamond\left(\xi \wedge \neg S_b\right)\right) \cdot \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) \\ & + \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \left(\mathbb{E}_{\mathcal{M}'}(\diamondplus\, s_b \mid \Diamond s_b) + \mathbb{E}_{\mathcal{M}, s_b}(\diamondplus\,\xi)\right). \end{aligned} \tag{FE}$$

The lemma claims that we can partition the set of paths satisfying $\Diamond\xi$ into the set of paths where ξ is reached *before* reaching S_2, and the set of paths where ξ is reached *when or after* reaching S_2. Each path in the latter set of paths is split into two sub-paths, and computation of sub-results is performed on the set of sub-paths.

Proof of Lemma 4.2. We start with proving (FP). We partition the set of paths $\Pi_{s_{init}}(\xi)$ into two sets. The first set contains all paths where ξ is reached before S_2 (including paths where S_2 is never reached). The second set contains all paths where ξ is reached earliest with reaching S_2. Note that for all paths visiting a state in S_2, the first state of this path that is in S_2 needs to be in S_b. Formally, all paths $\pi \in \mathit{Paths}$ visiting a state in S_2 are of the form $\pi = s_1 \ldots s_b s_{b+1} \ldots$, $b \geq 1$ such that $s_i \notin S_b$ for $i < b$, $s_b \in S_b$ and $s_j \in S_1 \cup S_2$ for

$j > b$. Thus we can further partition the set of paths visiting a state in S_2 by the boundary state $s_b \in S_b$. In the proof we use the following LTL-equivalence:

$$\Diamond \xi \equiv_{\mathcal{M}} \underbrace{(\neg S_b \,\mathrm{U}\, (\xi \wedge \neg S_b))}_{\text{“}\xi\text{ before }S_2\text{”}} \vee \underbrace{\left(\bigvee_{s_b \in S_b} \neg \xi \,\mathrm{U}\, (s_b \wedge \Diamond \xi) \right)}_{\text{“}S_2\text{ before or with }\xi\text{”}}$$

where $\equiv_{\mathcal{M}}$ denotes that all paths in $\mathit{FPaths}_{\mathcal{M},s_{init}}$ that satisfy the left-hand side formula also satisfy the right-hand side formula of the equivalence, and vice versa. In $\mathcal{M}'$, where all states in S_2 are absorbing, we have $(\neg S_b \,\mathrm{U}\, (\xi \wedge \neg S_b)) \equiv_{\mathcal{M}'} \Diamond(\xi \wedge \neg S_b)$.

All states in ξ are absorbing in $\mathcal{M}$, and thus for all $s_b \in S_b$ we have: $\neg\xi \,\mathrm{U}\, (s_b \wedge \Diamond\xi) \equiv_{\mathcal{M}} \Diamond s_b \wedge (\Diamond\xi)$, and with the same argument $\neg\xi \,\mathrm{U}\, s_b \equiv_{\mathcal{M}} \Diamond s_b$. Having this in mind, we retrieve:

$$\begin{aligned}
\Pr_{\mathcal{M}}(\Diamond\xi) &= \sum_{\pi \in \Pi(\xi)} \Pr(\pi) \\
&= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \sum_{\substack{\pi \in \Pi(\xi) \\ \pi \models \neg\xi \,\mathrm{U}\, (s_b \wedge \Diamond\xi)}} \Pr(\pi).
\end{aligned}$$

Each path in the inner sum index set is of the form $\pi = s_1 \dots s_b \dots s_k$ with $s_k \in \xi$, $s_i \notin \xi$ for $1 \leq i < k$. Note that, if $s_b \in \xi$, then $b = k$. We split the probability of each path into the product of the probabilities of the sub-paths $\pi_1 = s_1 \dots s_b$ and $\pi_2 = s_b \dots s_k$. For all paths π that reach S_2 via some state $s_b \in \xi$, π is split into sub-paths $\pi_1 = \pi$ and $\pi_2 = s_b$ with $\Pr_{\mathcal{M},s_b}(s_b) = 1$.

$$\begin{aligned}
\Pr_{\mathcal{M}}(\Diamond\xi) &= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \sum_{\substack{\pi_1 \in \Pi(s_b) \\ \pi_1 \models \neg\xi \,\mathrm{U}\, s_b}} \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_1) \cdot \Pr(\pi_2) \\
&= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \sum_{\substack{\pi_1 \in \Pi(s_b) \\ \pi_1 \models \neg\xi \,\mathrm{U}\, s_b}} \Pr(\pi_1) \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_2).
\end{aligned}$$

Using the LTL equivalence from above, we know that each path satisfying $\Diamond s_b$ also satisfies $\neg\xi \,\mathrm{U}\, s_b$, and vice versa. Thus, we can omit the condition $\pi \models \neg\xi \,\mathrm{U}\, s_b$ in the sum index.

$$\begin{aligned}
\Pr_{\mathcal{M}}(\Diamond\xi) &= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \sum_{\pi_1 \in \Pi(s_b)} \Pr_{\mathcal{M}}(\pi_1) \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr_{\mathcal{M},s_b}(\pi_2) \\
&= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \Pr_{\mathcal{M}}(\Diamond s_b) \cdot \Pr_{\mathcal{M},s_b}(\Diamond\xi) \\
&= \Pr_{\mathcal{M}'}\left(\Diamond\left(\xi \wedge \neg S_b\right)\right) + \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \Pr_{\mathcal{M},s_b}(\Diamond\xi).
\end{aligned}$$

The proof of (FE) follows the same schema. Recall from Section 2 that for any $s \in S$ and non-empty set of states $T \subseteq S$ the equation $\sum_{\pi \in \Pi_s(T)} \Pr(\pi) \cdot \mathit{rew}(\pi) = \mathbb{E}_{\mathcal{M},s}\left(\diamondplus T\right)$ holds only if $\Pr_{\mathcal{M},s}(\Diamond T) = 1$. If $\Pr_{\mathcal{M},s}(\Diamond T) < 1$, we can use

$$\sum_{\pi \in \Pi_s(T)} \Pr(\pi) \cdot \mathit{rew}(\pi) = \sum_{\pi \in \Pi_s(T)} \Pr(\pi) \cdot \mathit{rew}(\pi) \cdot \frac{\Pr_{\mathcal{M},s}(\Diamond T)}{\Pr_{\mathcal{M},s}(\Diamond T)}$$

$$= \mathbb{E}_{\mathcal{M},s}(\Diamond\!\!\!\!\!+\, T \mid \Diamond T) \cdot \Pr_{\mathcal{M},s}(\Diamond T).$$

This yields:

$$\begin{aligned}
\mathbb{E}_{\mathcal{M}}(\Diamond\!\!\!\!\!+\, \xi) &= \sum_{\pi \in \Pi(\xi)} \Pr(\pi) \cdot \mathit{rew}(\pi) \\
&= \sum_{\substack{\pi \in \Pi(\xi) \\ \pi \models \Box \neg S_b}} \Pr(\pi) \cdot \mathit{rew}(\pi) + \sum_{s_b \in S_b} \sum_{\substack{\pi \in \Pi(\xi) \\ \pi \models \Diamond s_b}} \Pr(\pi) \cdot \mathit{rew}(\pi) \\
&= \mathbb{E}_{\mathcal{M}'}\left(\Diamond\!\!\!\!\!+\, (\xi \wedge \neg S_b) \mid \Diamond\, (\xi \wedge \neg S_b)\right) \cdot \Pr_{\mathcal{M}'}\left(\Diamond\, (\xi \wedge \neg S_b)\right) \\
&\qquad + \sum_{s_b \in S_b} \sum_{\substack{\pi \in \Pi(\xi) \\ \pi \models \neg\xi \,\mathrm{U}\, (s_b \wedge \Diamond \xi) s_b}} \Pr(\pi) \cdot \mathit{rew}(\pi).
\end{aligned}$$

We have a closer look at the second summand.

$$\begin{aligned}
&\sum_{s_b \in S_b} \sum_{\substack{\pi \in \Pi(\xi) \\ \pi \models \neg\xi \,\mathrm{U}\, (s_b \wedge \Diamond \xi)}} \Pr(\pi) \cdot \mathit{rew}(\pi) \\
&= \sum_{s_b \in S_b} \sum_{\substack{\pi_1 \in \Pi(s_b) \\ \pi_1 \models \neg\xi \,\mathrm{U}\, s_b}} \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_1) \cdot \Pr(\pi_2) \cdot \left(\mathit{rew}\,(\pi_1) + \mathit{rew}\,(\pi_2)\right) \\
&= \sum_{s_b \in S_b} \sum_{\substack{\pi_1 \in \Pi(s_b) \\ \pi_1 \models \neg\xi \,\mathrm{U}\, s_b}} \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_1) \cdot \Pr(\pi_2) \cdot \mathit{rew}\,(\pi_1) \\
&\quad + \sum_{s_b \in S_b} \sum_{\substack{\pi_1 \in \Pi(s_b) \\ \pi_1 \models \neg\xi \,\mathrm{U}\, s_b}} \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_1) \cdot \Pr(\pi_2) \cdot \mathit{rew}\,(\pi_2) \\
&= \sum_{s_b \in S_b} \sum_{\pi_1 \in \Pi(s_b)} \Pr(\pi_1) \cdot \mathit{rew}\,(\pi_1) \cdot \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_2) \\
&\quad + \sum_{s_b \in S_b} \sum_{\pi_1 \in \Pi(s_b)} \Pr(\pi_1) \cdot \sum_{\pi_2 \in \Pi_{s_b}(\xi)} \Pr(\pi_2) \cdot \mathit{rew}\,(\pi_2) \\
&= \sum_{s_b \in S_b} \mathbb{E}_{\mathcal{M}'}(\Diamond\!\!\!\!\!+\, s_b \mid \Diamond s_b) \cdot \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \Pr_{\mathcal{M},s_b}(\Diamond \xi) \\
&\quad + \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \mathbb{E}_{\mathcal{M},s_b}(\Diamond\!\!\!\!\!+\, \xi) && (*) \\
&= \sum_{s_b \in S_b} \mathbb{E}_{\mathcal{M}'}(\Diamond\!\!\!\!\!+\, s_b \mid \Diamond s_b) \cdot \Pr_{\mathcal{M}'}(\Diamond s_b) + \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \mathbb{E}_{\mathcal{M},s_b}(\Diamond\!\!\!\!\!+\, \xi) && (**) \\
&= \sum_{s_b \in S_b} \Pr_{\mathcal{M}'}(\Diamond s_b) \cdot \left(\mathbb{E}_{\mathcal{M}'}(\Diamond\!\!\!\!\!+\, s_b \mid \Diamond s_b) + \mathbb{E}_{\mathcal{M},s_b}(\Diamond\!\!\!\!\!+\, \xi)\right).
\end{aligned}$$

The transformations (*) and (**) arise since $\Pr_{\mathcal{M},s_{init}}(\Diamond \xi) = 1$ and for all states $s_b \in S_b$: $\Pr_{\mathcal{M}}(\Diamond s_b) > 0$. Thus, we have $\Pr_{\mathcal{M},s_b}(\Diamond \xi) = 1$ for all $s_b \in S_b$. □

Remark 4.3 (Counter-based Adaption). *In the next section, we will choose S_1 and S_2 such that $\Pr_{\mathcal{M},t}(\Diamond S_1) = 0$ for all $t \in S_2$. Hence, for all $s_b \in S_b$, when setting s_b as initial state, states in S_1 are not reachable and thus can be removed. Furthermore, $\Pr_{\mathcal{M}'}(\Diamond s_b)$ and $\mathbb{E}_{\mathcal{M}'}(\Diamond\!\!\!\!\!+\, s_b \mid \Diamond s_b)$ are computed in the DTMC $\mathcal{M}'$ where all states in S_b are absorbing and thus states in $S_2 \setminus S_b$ are not reachable.*

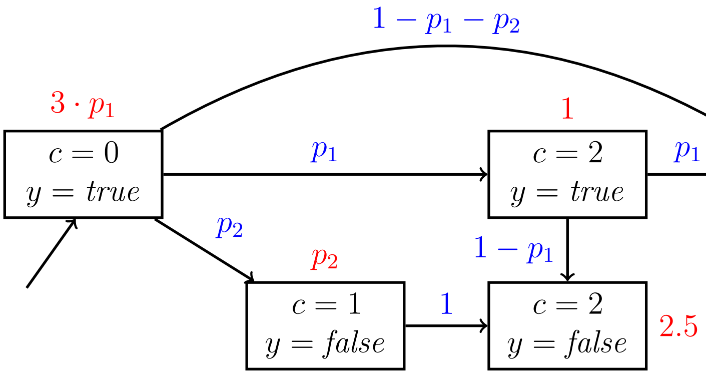

Figure 4.2: A DTMC with counter c. Probabilities are denoted by blue transition annotations, reward assignments are red.

4.2 Counters

The basis of counter-based simplification is: a counter. A counter is a variable of the underlying program graph whose value never decreases. In this section we give the formal definition of counters, and show how a counter induces a partitioning of the state set such that factorization as defined in the previous section is applicable.

After defining counters, we transform the DTMC such that states that are reached by transitions that update the counter can be identified. These states assume the role of boundary states, and we obtain the base schema of counter-based factorization.

In the rest of this chapter we fix a probabilistic program graph $\mathcal{P} = (\mathit{LV}, \mathit{Act}, \hookrightarrow)$ over a non-empty set of variables Vars, and an initial location $\mathfrak{l}_{\mathit{init}}$. Furthermore, we fix a reward structure rew over $\mathcal{P}$. The induced DTMC of $\mathcal{P}$ is denoted by $\mathcal{M} = (S, P, \mathsf{L})$, and $s_{\mathit{init}} = \mathfrak{l}_{\mathit{init}}$.

Definition 4.4 (Counters). *A* counter *is a variable $c \in \mathit{Vars}$ of the probabilistic program graph $\mathcal{P}$ such that for all paths $\pi \in \mathit{Paths}(\mathcal{M})$: $\pi \models \Box \bigwedge_{x \in D(c)} (c = x \implies \Box c \geq x)$, where $D(c)$ denotes the domain of c.*

We say that c is a counter of $\mathcal{M}$, if it is a counter of the underlying program graph $\mathcal{P}$. In the further, let c be a counter of $\mathcal{M}$ with minimal value 0 and maximal value n.

Example 4.5. *Consider the DTMC $\mathcal{M}$ of Figure 4.2, arising from a probabilistic program graph with variables c and y, and two parameters p_1 and p_2. Variable c is a counter, since every transition either updates the counter or leaves its value unchanged. The maximal value of c is $n = 4$. We will use this DTMC and $s_{\mathit{init}} = (c = 0, y = \mathit{true})$ as a running example, and exemplarily compute the rational function for $\mathbb{E}_{\mathcal{M}}(\Diamond((y = \mathit{false} \wedge c = 2) \vee c = 4)) = p_2^2 + 6p_1^2 - 2p_1 - 6p_2 + 6$ using factorization.*

Factorization makes use of a partitioning of the state space and the concept of boundary states. In counter-based factorization, the partitioning follows the counter updates. Thus, the boundary states are states where the counter is “freshly updated”. To uniquely identify these states, we transform the DTMC $\mathcal{M}$ such that in all paths of the DTMC all states that are reached by a transition updating the counter are flagged. This flagged DTMC $\mathcal{M}$

arises from $\mathcal{M}$ by duplicating the state space, flagging each new state, and transforming the transition probability function such that each transition that updates the counter reaches the flagged copy of the original transition's target.

Definition 4.6 (Counter-Update-Flagged DTMC). *Let* $S_c = \{(s, f) \mid s \in S\}$ *be a set of states that contains a flagged copy of each state in* S*. Let* $s{\downarrow_{\mathcal{M}}} = s$ *for all* $s \in S$ *and* $(s, f){\downarrow_{\mathcal{M}}} = s$ *for all* $(s, f) \in S_c$*. The counter-update-flagged DTMC of* $\mathcal{M}$ *is* $\mathcal{M}_f = (S_f, P_f, \mathsf{L}_f)$ *with:*

- $S_f = S \cup S_c$.
- *For all* $s \in S_f$, $t \in S$, $(t, f) \in S_c$*:*

$$P_f(s,t) = \begin{cases} P(s{\downarrow_{\mathcal{M}}}, t) & \text{if } s{\downarrow_{\mathcal{M}}}(c) = t(c) \\ 0 & \text{if } s{\downarrow_{\mathcal{M}}}(c) \neq t(c) \end{cases} \qquad \text{and}$$

$$P_f(s,(t,f)) = \begin{cases} P(s{\downarrow_{\mathcal{M}}}, t) & \text{if } s{\downarrow_{\mathcal{M}}}(c) \neq t(c) \\ 0 & \text{if } s{\downarrow_{\mathcal{M}}}(c) = t(c). \end{cases}$$

- *For all* $s \in S$*:* $\mathsf{L}_f(s) = \mathsf{L}(s)$ *and for all* $(s, f) \in S_c : \mathsf{L}_f(s, f) = \mathsf{L}(s) \cup f$.

The states in S_c *are called* counter states*. We write* $s(f) = \mathit{true}$ *for all states* $s \in S_c$ *and* $s(f) = \mathit{false}$ *for all* $s \in S$.

Corollary 4.7. *Let* $\xi_f = \xi \cup \{(s, f) \in S_c \mid s \in \xi\}$ *be the flagged version of* ξ*. Let* $\mathit{rew}_f \colon S_f \to \mathbb{Q}^{\geq 0}$ *with* $\mathit{rew}_f(s) = \mathit{rew}(s{\downarrow_{\mathcal{M}}})$ *for all* $s \in S_f$ *be the flagged version of rew. We have* $\mathrm{Pr}_{\mathcal{M},s_{init}}(\Diamond\xi) = \mathrm{Pr}_{\mathcal{M}_f,s_{init}}(\Diamond\xi_f)$ *and* $\mathbb{E}^{\mathit{rew}}_{\mathcal{M},s_{init}}(\oplus\,\xi) = \mathbb{E}^{\mathit{rew}_f}_{\mathcal{M},s_{init}}(\oplus\,\xi_f)$.

Example 4.8 (Counter-Update-Flagged DTMC). *When flagging the DTMC of example 4.5, (see Figure 4.3) all transitions from* $(c = 0, y = \mathit{true})$ *change the counter value and thus lead to a flagged copy of the original target state. Furthermore, there is a transition from the flagged copy* $(c = 1, y = \mathit{false}, f)$ *to the flagged state* $(c = 2, y = \mathit{false}, f)$*. The original transition from* $(c = 2, y = \mathit{true})$ *to* $(c = 2, y = \mathit{false})$ *does not update the counter, thus there is a transition from the flagged copy* $(c = 2, y = \mathit{true}, f)$ *to the unflagged state* $(c = 2, y = \mathit{false})$*. The original transition from* $(c = 2, y = \mathit{true})$ *to* $(c = 3, y = \mathit{true})$ *updates the counter, and thus leads from the flagged copy* $(c = 2, y = \mathit{true}, f)$ *to the flagged copy* $(c = 3, y = \mathit{true}, f)$*.*

In the original DTMC there is a self loop in state $(c = 3, y = \mathit{true})$*. Since this self loop does not update the counter, it is still present in the unflagged state* $(c = 3, y = \mathit{true})$ *in* $\mathcal{M}$*. Furthermore, there is a transition from the flagged copy* $(c = 3, y = \mathit{true}, f)$ *to the unflagged state* $(c = 3, y = \mathit{true})$*.*

The original transition from $(c = 3, y = \mathit{true})$ *to* $(c = 4, y = \mathit{true})$ *is present in both states in* $\mathcal{M}$*, the original state* $(c = 3, y = \mathit{true})$ *and its flagged copy. Since these transitions update the counter, they lead to the flagged state* $(c = 4, y = \mathit{true}, f)$*.*

States $(c = 2, y = \mathit{false})$ *and* $(c = 4, y = \mathit{true})$ *are absorbing states in the original DTMC, thus, in* $\mathcal{M}$ *these states as well as their flagged copies are absorbing, too.*

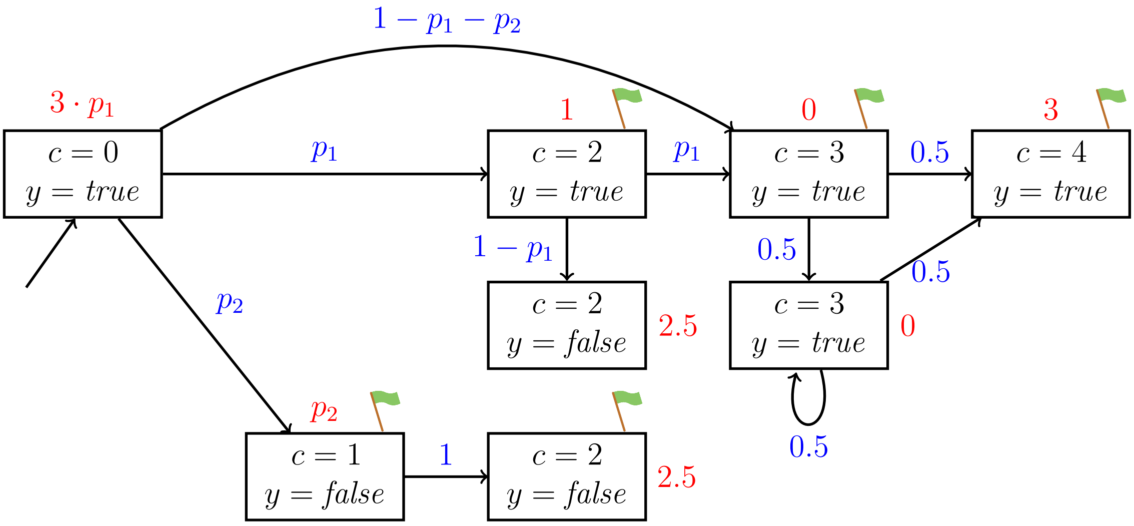

Figure 4.3: The flagged DTMC of Example 4.5. States in S_c are marked with a green flag.

In the further, we omit the index f and assume a DTMC $\mathcal{M}$ that is already flagged. We now apply the factorization approach from the previous section to the counter-flagged DTMC, using a subset of the counter states as boundary set. This subset is the set of states that for some path in the DTMC $\mathcal{M}$ is reached with the first counter update in this path.

Theorem 4.9 (Counter-based Factorization). *Let $\mathcal{M}^1 = (S^1, P^1, \mathsf{L}^1)$ be the DTMC arising from $\mathcal{M}$ by making all counter states absorbing. Let $S_1 = S^1 \setminus S_c$ be the set of states in $\mathcal{M}$ that can be reached from the initial state without updating the counter and $S_2 = S \setminus S_1$ be all other states in $\mathcal{M}$. Let $S_c^1 = S^1 \cap S_c$ be the set of boundary states. Then:*

$$\Pr_{\mathcal{M}}(\lozenge\xi) = \Pr_{\mathcal{M}^1}\left(\lozenge\left(\xi \land \neg S_c^1\right)\right) + \sum_{s_b \in S_c^1} \Pr_{\mathcal{M}^1}(\lozenge s_b) \cdot \Pr_{\mathcal{M},s_b}(\lozenge\xi).$$

Let $\Pr_{\mathcal{M}}(\lozenge\xi) = 1$. Then:

$$\begin{aligned}\mathbb{E}_{\mathcal{M}}(\oplus\xi) &= \mathbb{E}_{\mathcal{M}^1}\left(\oplus\left(\xi \land \neg S_c^1\right) \mid \lozenge\left(\xi \land \neg S_c^1\right)\right) \cdot \Pr_{\mathcal{M}^1}\left(\lozenge\left(\xi \land \neg S_c^1\right)\right) \\ &+ \sum_{s_b \in S_c^1} \Pr_{\mathcal{M}^1}(\lozenge s_b) \cdot \left(\mathbb{E}_{\mathcal{M}^1}(\oplus s_b \mid \lozenge s_b) + \mathbb{E}_{\mathcal{M},s_b}(\oplus\xi)\right)\end{aligned}$$

Proof of Theorem 4.9. The proof follows directly from Lemma 4.2. □

The factorization approach enables the computation of global probabilistic reachability properties and expected accumulated rewards by computing local properties in two *factors* of the DTMCs: one factor comprising only the states of $\mathcal{M}$ that are reached before or with the first counter update, and one factor comprising the states that are reached with or after the first update. Computation of properties in the latter can be further simplified by threating the (new) initial state (which is one of the states in S_c^1) as unflagged, retrieving a new set of boundary states (which is now the set of counter states where the counter is updated for the second time) and using counter-based factorization on this sub-DTMC.

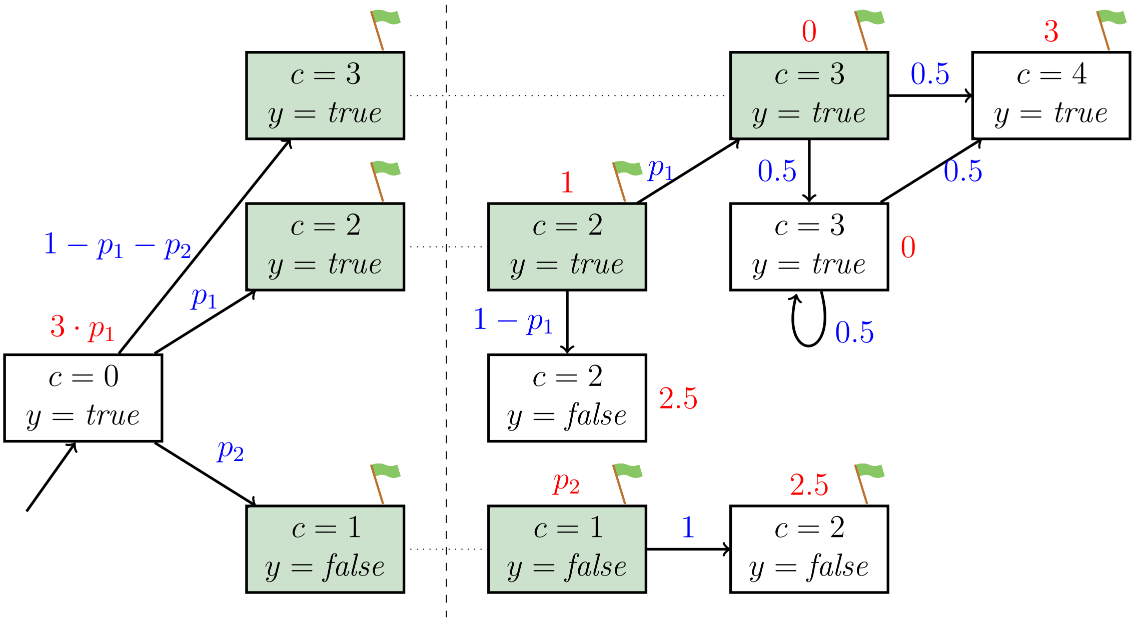

Figure 4.4: Factors $\mathcal{M}^1$ (left) and $\mathcal{M}^r$ (right) arising from applying factorization to the flagged DTMC $\mathcal{M}$ of Example 4.5. States in S_c^1 have a green shade and appear in both factors.

This approach can be applied recursively until obtaining a factor that does not contain any counter states except for the respective initial state.

Example 4.10. *We apply factorization to the flagged DTMC $\mathcal{M}$ in Figure 4.3. We use factorization twice, and thus retrieve three factors $\mathcal{M}^1$, $\mathcal{M}^2$, and $\mathcal{M}^3$. When factorizing the first time, we use a set of boundary states S_c^1 and factorize $\mathcal{M}$ into two parts, $\mathcal{M}^1$ and $\mathcal{M}^r$. Latter one will be factorized again, using boundary states S_c^2, to retrieve factors $\mathcal{M}^2$, and $\mathcal{M}^3$.*

The factors arising from applying counter-based factorization to the flagged DTMC from Figure 4.3 are depicted in Figure 4.4. The boundary states S_c^1 are the first flagged states that can be reached from the initial state $(c = 0, y = true)$: $(c = 3, y = true, f)$, $(c = 2, y = true, f)$, and $(c = 1, y = false, f)$. These states are made absorbing in the first factor ($\mathcal{M}^1$) of the DTMC. The second factor, $\mathcal{M}^r$, contains the states in S_c^1 and all states that are reachable from S_c^1.

To compute the expected accumulated reward $\mathbb{E}_{\mathcal{M}}(\diamondsuit(y = false \vee c = 4))$, we calculate the probability $\Pr_{\mathcal{M}^1}(\Diamond\xi \wedge \neg S_c^1)$, which is 0. Thus, there is no need to compute the expected reward $\mathbb{E}_{\mathcal{M}^1}(\diamondsuit(\xi \wedge \neg S_c^1) \mid \Diamond\xi \wedge \neg S_c^1)$.

Furthermore, for all boundary states $s_b \in S_c^1$ the probability of reaching this state from the initial state needs to be computed. We have

$$\Pr_{\mathcal{M}^1}(\Diamond(c = 3, y = true, f)) = 1 - p_1 - p_2,$$
$$\Pr_{\mathcal{M}^1}(\Diamond(c = 2, y = true, f)) = p_1,$$
$$\Pr_{\mathcal{M}^1}(\Diamond(c = 1, y = false, f)) = p_2.$$

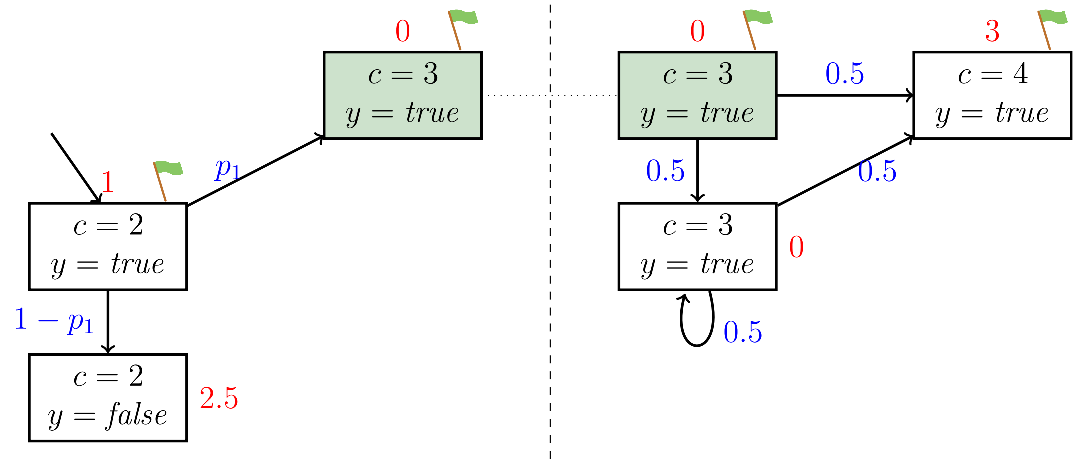

Figure 4.5: Factors $\mathcal{M}^2$ (left) and $\mathcal{M}^3$ (right) arising from applying factorization to $\mathcal{M}^r$, when treating $(c = 2, y = \mathit{true}, f)$ as initial state. States in S_c^2 have a green shade and appear in both factors.

Also for all states $s_b \in S_c^1$, we to compute the conditional expected reward until reaching s_b. This is

$$
\begin{aligned}
&\mathbb{E}_{\mathcal{M}^1}(\diamondplus(c = 3, y = \mathit{true}, f) \mid \Diamond(c = 3, y = \mathit{true}, f)) = 3 \cdot p_1 \\
&\mathbb{E}_{\mathcal{M}^1}(\diamondplus(c = 2, y = \mathit{true}, f) \mid \Diamond(c = 2, y = \mathit{true}, f)) = 3 \cdot p_1, \\
&\mathbb{E}_{\mathcal{M}^1}(\diamondplus(c = 1, y = \mathit{false}, f) \mid \Diamond(c = 1, y = \mathit{false}, f)) = 3 \cdot p_1.
\end{aligned}
$$

Now, for all states $s_b \in S_c^1$, we compute $\mathbb{E}_{\mathcal{M}^r, s_b}(\diamondplus\,\xi \mid \Diamond\xi)$. For $s_b = (c = 3, y = \mathit{true}, f)$, this value is 6. For $s_b = (c = 1, y = \mathit{false}, f)$ it is p_2. To compute the expected reward $\mathbb{E}_{\mathcal{M}^r,(c=2,y=\mathit{true},f)}(\diamondplus\,\xi \mid \Diamond\xi)$, we apply factorization to $\mathcal{M}^r$. Figure 4.5 depicts the factors arising from factorizing $\mathcal{M}^r$, when setting $(c = 2, y = \mathit{true}, f)$ as initial state. The new set of boundary states S_c^2 is the single state $s_b = (c = 3, y = \mathit{true}, f)$. We calculate in the factor $\mathcal{M}^2$ the probability and expected accumulated reward of reaching a state in ξ. The only state in ξ in this factor is $(c = 2, y = \mathit{false})$, and we obtain

$$
\begin{aligned}
&\Pr_{\mathcal{M}^2,(c=2,y=\mathit{true},f)}(\Diamond\xi) = 1 - p_1 \qquad \textit{and} \\
&\mathbb{E}_{\mathcal{M}^2,(c=2,y=\mathit{true},f)}(\diamondplus\,\xi \mid \Diamond\xi) = 1.
\end{aligned}
$$

Furthermore, the probability and conditional expected accumulated reward of reaching the single boundary state are:

$$
\begin{aligned}
&\Pr_{\mathcal{M}^2,(c=2,y=\mathit{true},f)}(\Diamond(c = 3, y = \mathit{true}, f)) = p_1 \qquad \textit{and} \\
&\mathbb{E}_{\mathcal{M}^2,(c=2,y=\mathit{true},f)}(\diamondplus(c = 3, y = \mathit{true}, f) \mid \Diamond(c = 3, y = \mathit{true}, f)) = 1.
\end{aligned}
$$

The expected accumulated reward of reaching ξ in the factor $\mathcal{M}^3$ from $(c = 3, y = \mathit{true}, f)$ was already computed in $\mathcal{M}^r$ (the result is 6) and can be re-used. Using the results from factors $\mathcal{M}^2$ and $\mathcal{M}^3$, we obtain

$$
\mathbb{E}_{\mathcal{M}^r,(c=2,y=\mathit{true},f)}(\diamondplus\,\xi \mid \Diamond\xi) = (1 - p_1) \cdot 1 + p_1 \cdot 1 + p_1 \cdot 6 = 6p_1 + 1,
$$

and finally:

$$\begin{aligned}
\mathbb{E}_{\mathcal{M}}(\diamondplus\, \xi) &= \sum_{s_b \in S_c^1} \Pr_{\mathcal{M}^1}(\Diamond s_b) \cdot (\mathbb{E}_{\mathcal{M}^1}(\diamondplus\, s_b \mid \Diamond s_b) + \mathbb{E}_{\mathcal{M}^r, s_b}(\diamondplus\, \xi)) \\
&= (1 - p_1 - p_2) \cdot (3p_1 + 6) \quad + \quad p_1 \cdot (3p_1 + (6p_1 + 1)) \quad + \quad p_2 \cdot (3p_1 + p_2) \\
&= p_2^2 + 6p_1^2 - 2p_1 - 6p_2 + 6.
\end{aligned}$$

4.3 Factorization for Simple Counters

In this section we define simple counters (counters, where each counter update increases the counter by the same value), and give the factorization approach for simple counters. Then, we dive into the special case of simple counters being observing (the counter value does not influence the model behavior), and show how this scenario simplifies factorization.

Definition 4.11 (Simple Counters). *A counter c of $\mathcal{P}$ is* simple*, if there is a natural number $x > 0$ such that for all states $s, t \in S$ with $P(s,t) > 0$ we have that either $s(c) = t(c)$ or $s(c) = t(c) - x$. The counter c is a simple counter with update value x. A counter that is not simple is* multivariant*.*

Each PPG having a simple counter with update value $x \in \mathbb{N}$, $x > 0$, can be transformed into a PPG having a simple counter with update value 1, by dividing each occurrence of the counter value (in the evaluations, updates, reward structure, etc.) by x. In the further, we assume the counter c to be simple with update value 1. Furthermore a DTMC with multivariant counter c can be transformed such that the counter is simple, by splitting each transition updating the counter into a chain of transitions where each new transition updates the counter by 1 with probability 1.

The factors arising from factorization for simple counters can be defined using the counter values.

Definition 4.12 (Factors and Counter States). *Let $0 \leq i \leq n$. The DTMC $\mathcal{M}^i = (S^i, P^i, \mathsf{L}^i)$ with*

- $S^i = \{s \in S \mid s(c) = i\} \cup \{s \in S_c \mid s(c) = i + 1\}$,
- $P^i(s,t) = \begin{cases} P(s,t) & \text{if } s,t \in S^i \\ 0 & \text{otherwise} \end{cases}$
- $\mathsf{L}^i(s) = \mathsf{L}(s)$ *for all* $s \in S^i$

is called a factor *of $\mathcal{M}$. The states in $S_c^i = \{s \in S_c \mid s(c) = i\}$ are called counter states of factor i.*

The DTMC in Figure 4.6 depicts the factors arising from simple counters, and the counter states that are reached with an counter update. For each i, the factor $\mathcal{M}^i$ is the part of the DTMC $\mathcal{M}$ that contains all states between the counter states of $\mathcal{M}$ with value i and

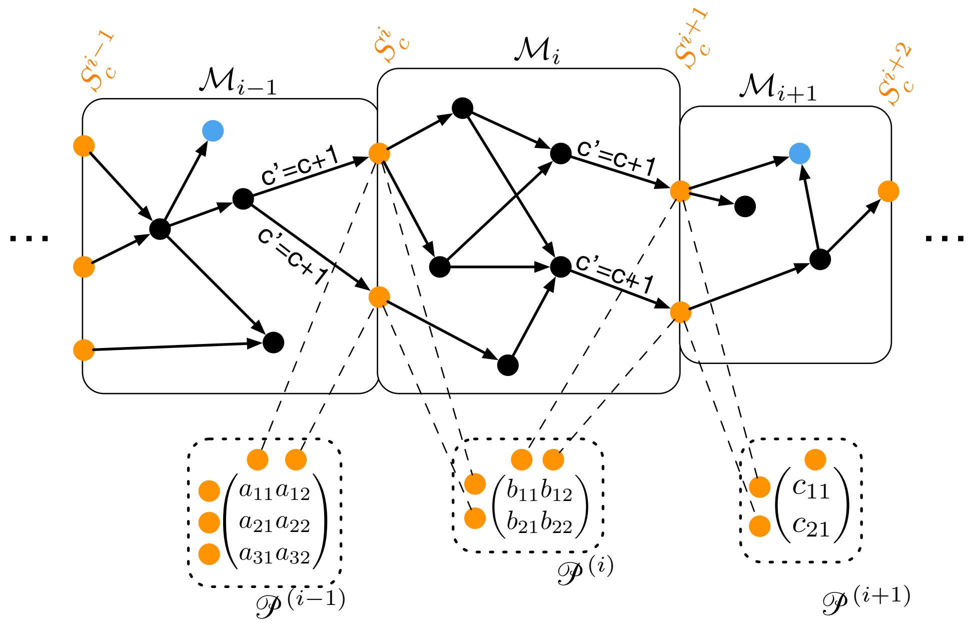
S_c^{i-1}
$\mathcal{M}_{i-1}$
S_c^{i}
$\mathcal{M}_{i}$
S_c^{i+1}
$\mathcal{M}_{i+1}$
S_c^{i+2}
c'=c+1
c'=c+1
c'=c+1
c'=c+1
...
...
$\begin{pmatrix} a_{11} a_{12} \\ a_{21} a_{22} \\ a_{31} a_{32} \end{pmatrix}$
$\begin{pmatrix} b_{11} b_{12} \\ b_{21} b_{22} \end{pmatrix}$
$\begin{pmatrix} c_{11} \\ c_{21} \end{pmatrix}$
$\mathscr{P}^{(i-1)}$
$\mathscr{P}^{(i)}$
$\mathscr{P}^{(i+1)}$

Figure 4.6: Factors $\mathcal{M}^{i-1}$, $\mathcal{M}^{i}$, and $\mathcal{M}^{i+1}$ arising from a simple counter c. States in ξ are marked blue, and counter states are orange. Below the Markov chain, the size of the matrices $\mathscr{P}^{(j)}$ is depicted. For the depicted Markov chain, $\mathscr{P}^{(i-1)}$ has three rows and two columns, since $|S_c^{i-1}| = 3$ and $|S_c^i| = 2$.

the counter states with value $i+1$. Thus, all states s in factor $\mathcal{M}^i$ we have $s(c) = i$ or $s(c) = i+1$. The factor $\mathcal{M}^0$ comprises states where $c = 0$ and the set of counter state S_c^1, i.e., flagged states with counter value 1. $\mathcal{M}^1$ then comprises states "between the flagged states with value 1 and value 2", i.e., all states where the counter value is 1 plus the flagged states in S_c^2. Note that factor $\mathcal{M}^n$ only comprises states with counter value n, since there are no flagged states with counter value $n+1$.

Proposition 4.13 (Well-Definedness of $\mathcal{M}^i$). *For each $0 \leq i < n$, the DTMC $\mathcal{M}^i$ is well-defined, i.e, for all states $s \in S^i$: $P^i(s,t) = 0$ for all $t \in S^i$ or $\sum_{t \in S^i} P^i(s,t) = 1$.*

Proof. Let $s \in S^i$. Then $s(c) = i$ or $s(c) = i+1$. We show: $s(c) = i \implies \sum_{t \in S^i} P^i(s,t) = 1$ and $s(c) = i+1 \implies P^i(s,t) = 0$ for all $t \in S^i$.

Case $s(c) = i$: For all states $t \in S$ with $P(s,t) > 0$ we have: Either $t(c) = i$ and thus $t \in S^i$. Or $t(c) = i+1$, since we assumed c to be a simple counter with update value 1. Then, t is reached from s by a transition that updates the counter, thus $t \in S_c$ and $t \in S^i$. Therefore, all states t that are reachable from s in $\mathcal{M}$ are reachable from s in $\mathcal{M}^i$. Hence, $\sum_{t \in S^i} P^i(s,t) = \sum_{t \in S} P(s,t) = 1$.

Case $s(c) = i+1$: Then $s \in S_c$. We show that there is no state in $t \in S^i$ such that $P^i(s,t) > 0$. All states $t \in S^i$ with $t(c) = i$ are not reachable from s, since c is a counter. All states $t \in S^i$ with $t(c) = i+1$ are counter states, and hence are only reachable from states s' with counter value $s'(c) \leq i$. Since $s(c) = i+1$, t can not be reached from s, and hence $P^i(s,t) = 0$. $\square$

Lemma 4.14. *Let* $s_0 = s_{\mathit{init}}$, $S_c^0 = \{s_0\}$, *and for all* $0 \le i < n$, *let*

$$\begin{aligned}
\mathscr{P}^{(i)} &= \left(\Pr_{\mathcal{M}^i,s}(\Diamond t)\right)_{s\in S_c^i, t\in S_c^{i+1}}\,,\\
\mathfrak{p}^{(i)} &= \left(\Pr_{\mathcal{M}^i,s}\left(\Diamond\left(\xi \wedge \neg S_c^{i+1}\right)\right)\right)_{s\in S_c^i}\,,\\
\mathfrak{p}^{(n)} &= \left(\Pr_{\mathcal{M}^n,s}\left(\Diamond \xi\right)\right)_{s\in S_c^n}\,,\\
\mathscr{E}^{(i)} &= \left(\mathbb{E}_{\mathcal{M}^i,s}(\diamondplus t \mid \Diamond t)\cdot \Pr_{\mathcal{M}^i,s}(\Diamond t)\right)_{s\in S_c^i, t\in S_c^{i+1}}\,,\\
\mathfrak{e}^{(i)} &= \left(\mathbb{E}_{\mathcal{M}^i,s}\left(\diamondplus\left(\xi\wedge\neg S_c^{i+1}\right) \mid \Diamond\left(\xi\wedge\neg S_c^{i+1}\right)\right)\cdot \Pr_{\mathcal{M}^i,s}\left(\Diamond\left(\xi\wedge\neg S_c^{i+1}\right)\right)\right)_{s\in S_c^i}\,,
\end{aligned}$$

and

$$\mathfrak{e}^{(n)} = \left(\mathbb{E}_{\mathcal{M}^n,s}\left(\diamondplus \xi\right)\right)_{s\in S_c^n}\,.$$

Then:

$$\Pr_{\mathcal{M}}(\Diamond\xi) = \sum_{i=0}^{n}\left(\prod_{j=0}^{i-1}\mathscr{P}^{(j)}\cdot\mathfrak{p}^{(i)}\right)$$

and if $\Pr_{\mathcal{M}}(\Diamond\xi) = 1$, *then*

$$\mathbb{E}_{\mathcal{M}}(\diamondplus\xi) = \sum_{i=1}^{n}\sum_{s\in S_c^i}\left(\prod_{j=0}^{i-2}\mathscr{P}^{(j)}\cdot\mathscr{E}^{(i-1)}\right)_s + \sum_{i=0}^{n}\left(\prod_{j=0}^{i-1}\mathscr{P}^{(j)}\cdot\mathfrak{e}^{(i)}\right).$$

The lemma arises from recursively applying factorization to the DTMC $\mathcal{M}$, as outlined in the previous section.

Proof. Choosing $S_1 = S^0 \setminus S_c^1$ and $S_2 = \bigcup_{1\le i\le n} S^i$ as partitioning for factorization yields the set of boundary states $S_b = S_c^1$ and $\mathcal{M}' = \mathcal{M}^0$. Applying (FP) yields:

$$\Pr_{\mathcal{M}}(\Diamond\xi) = \Pr_{\mathcal{M}^0}\left(\Diamond\left(\xi\wedge\neg S_c^1\right)\right) + \sum_{s_1\in S_c^1}\Pr_{\mathcal{M}^0}(\Diamond s_1)\cdot\Pr_{\mathcal{M},s_1}(\Diamond\xi).$$

Applying (FP) recursively to $Pr_{\mathcal{M},s_i}(\Diamond\xi)$ (choosing $S_1 = S^i\setminus S_c^{i+1}$ and $S_2 = \bigcup_{i<j\le n} S^j$, and thus $S_b = S_c^{i+1}$) yields:

$$\begin{aligned}
\Pr_{\mathcal{M}}(\Diamond\xi) = {} & \Pr_{\mathcal{M}^0}\left(\Diamond\left(\xi\wedge\neg S_c^1\right)\right)\\
&+ \sum_{s_1\in S_c^1}\Bigg(\Pr_{\mathcal{M}^0}(\Diamond s_1)\cdot\Big(\Pr_{\mathcal{M}^1,s_1}\left(\Diamond\left(\xi\wedge\neg S_c^2\right)\right)\\
&\qquad + \sum_{s_2\in S_c^2}\Pr_{\mathcal{M}^1,s_1}(\Diamond s_2)\cdot\Pr_{\mathcal{M}^2,s_2}(\Diamond\xi)\Big)\Bigg)\\
& \dots\\
= {} & \Pr_{\mathcal{M}^0}\left(\Diamond\left(\xi\wedge\neg S_c^1\right)\right)\\
&+ \sum_{s_1\in S_c^1}\Bigg(\Pr_{\mathcal{M}^0}(\Diamond s_1)\cdot\Big(\Pr_{\mathcal{M}^1,s_1}\left(\Diamond\left(\xi\wedge\neg S_c^2\right)\right)\\
&\qquad + \sum_{s_2\in S_c^2}\Pr_{\mathcal{M}^1,s_1}(\Diamond s_2)\cdot\Big(\Pr_{\mathcal{M}^2,s_2}\left(\Diamond\left(\xi\wedge\neg S_c^3\right)\right)
\end{aligned}$$

$$
\begin{aligned}
&\quad \ldots \\
&\qquad\qquad + \sum_{s_n \in S_c^n} \Pr_{\mathcal{M}^{n-1}, s_{n-1}}(\Diamond s_n) \cdot \Pr_{\mathcal{M}^n, s_n}(\Diamond \xi) \Big) \Big) \Bigg) \\
&= \Pr_{\mathcal{M}^0}\left(\Diamond\left(\xi \wedge \neg S_c^1\right)\right) \\
&\quad + \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Pr_{\mathcal{M}^1, s_1}\left(\Diamond\left(\xi \wedge \neg S^2\right)\right) \\
&\quad + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Pr_{\mathcal{M}^1, s_1}(\Diamond s_2) \cdot \Pr_{\mathcal{M}^2, s_2}\left(\Diamond\left(\xi \wedge \neg S_c^3\right)\right) \\
&\quad \ldots \\
&\quad + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \cdots \sum_{s_n \in S_c^n} \Pr_{\mathcal{M}^0}(\Diamond s_1) \ldots \cdot \Pr_{\mathcal{M}^{n-1}, s_{n-1}}(\Diamond s_n) \cdot \Pr_{\mathcal{M}^n, s_n}(\Diamond \xi) \\
&= \mathfrak{p}_{s_0}^{(0)} \\
&\quad + \sum_{s_1 \in S_c^1} \mathscr{P}_{s_0, s_1}^{(0)} \cdot \mathfrak{p}_{s_1}^{(1)} \\
&\quad + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \mathscr{P}_{s_0, s_1}^{(0)} \cdot \mathscr{P}_{s_1, s_2}^{(1)} \cdot \mathfrak{p}_{s_2}^{(2)} \\
&\quad \ldots \\
&\quad + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \cdots \sum_{s_n \in S_c^n} \mathscr{P}_{s_0, s_1}^{(0)} \ldots \cdot \mathscr{P}_{s_{n-1}, s_n}^{(n-1)} \cdot \mathfrak{p}_{s_n}^{(n)}.
\end{aligned}
$$

Now observe that for all $j \leq n$ and for all $s_j \in S_c^j$: $\sum_{s_1 \in S_c^1} \cdots \sum_{s_j \in S_c^j} \mathscr{P}_{s_0, s_1}^{(0)} \cdot \ldots \cdot \mathscr{P}_{s_j, s_{j+1}}^{(j)} = \left(\mathscr{P}^{(0)} \cdot \ldots \cdot \mathscr{P}^{(j)}\right)_{s_0, s_{j+1}}$. The product of the matrices is well-defined, since for all 0 $\leq$i < j: matrix $\mathscr{P}^{(i)}$ has size $|S_c^i| \times |S_c^{i+1}|$, and matrix $\mathscr{P}^{(i+1)}$ has size $|S_c^{i+1}| \times |S_c^{i+2}|$ (cf. Figure 4.6).

$$
\begin{aligned}
\Pr_{\mathcal{M}}(\Diamond \xi) &= \mathfrak{p}_{s_0}^{(0)} \\
&\quad + \sum_{s_1 \in S_c^1} \mathscr{P}_{s_0, s_1}^{(0)} \cdot \mathfrak{p}_{s_1}^{(1)} \\
&\quad + \sum_{s_2 \in S_c^2} \left(\mathscr{P}^{(0)} \cdot \mathscr{P}^{(1)}\right)_{s_0, s_2} \cdot \mathfrak{p}_{s_2}^{(2)} \\
&\quad \ldots \\
&\quad + \sum_{s_n \in S_c^n} \left(\mathscr{P}^{(0)} \cdot \ldots \cdot \mathscr{P}^{(n-1)}\right)_{s_0, s_n} \cdot \mathfrak{p}_{s_n}^{(n)} \\
&= \sum_{i=0}^{n} \sum_{s_i \in S_c^i} \left(\prod_{j=0}^{i-1} \mathscr{P}^{(j)}\right)_{s_0, s_i} \cdot \mathfrak{p}_{s_i}^{(i)} \\
&= \sum_{i=0}^{n} \left(\prod_{j=0}^{i-1} \mathscr{P}^{(j)} \cdot \mathfrak{p}^{(i)}\right).
\end{aligned}
$$

The last step arises from the fact that $\mathscr{P}^{(0)}$ has size $1 \times |S_c^1|$, and for all $0 \leq i \leq n$ $\mathfrak{p}^{(i)}$ has size $|S_c^i| \times 1$. Thus, for each summand the matrix product results in a single value.

The proof for expected accumulated rewards follows the same schema.

$$\begin{aligned}
\mathbb{E}_{\mathcal{M}}(\diamondplus\,\xi) =& \mathbb{E}_{\mathcal{M}^0}\left(\diamondplus\left(\xi \wedge \neg S_c^1\right) \mid \Diamond\left(\xi \wedge \neg S_c^1\right)\right) \cdot \Pr_{\mathcal{M}^0}\left(\Diamond\left(\xi \wedge \neg S_c^1\right)\right)\\
&+ \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \left(\mathbb{E}_{\mathcal{M}^0}(\diamondplus\, s_1 \mid \Diamond s_1) + \mathbb{E}_{s_1}(\diamondplus\,\xi)\right).
\end{aligned}$$

Recursive application and using the definition of $\mathfrak{e}^{(i)}$ yields

$$\begin{aligned}
\mathbb{E}_{\mathcal{M}}(\diamondplus\,\xi) =& \mathfrak{e}_{s_0}^{(0)}\\
&+ \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Bigg(\mathbb{E}_{\mathcal{M}^0}(\diamondplus\, s_1 \mid \Diamond s_1) + \mathfrak{e}_{s_1}^{(1)}\\
&\quad + \sum_{s_2 \in S_c^2} \Pr_{\mathcal{M}^1}(\Diamond s_2) \cdot \left(\mathbb{E}_{\mathcal{M}^1, s_1}(\diamondplus\, s_2 \mid \Diamond s_2) + \mathbb{E}_{s_2}(\diamondplus\,\xi)\right) \Bigg)\\
&\dots\\
=& \mathfrak{e}_{s_0}^{(0)}\\
&+ \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Bigg(\mathbb{E}_{\mathcal{M}^0}(\diamondplus\, s_1 \mid \Diamond s_1) + \mathfrak{e}_{s_1}^{(1)}\\
&\quad + \sum_{s_2 \in S_c^2} \Pr_{\mathcal{M}^1}(\Diamond s_2) \cdot \Big(\mathbb{E}_{\mathcal{M}^1, s_1}(\diamondplus\, s_2 \mid \Diamond s_2) + \mathfrak{e}_{s_2}^{(2)}\\
&\quad \dots\\
&\qquad + \sum_{s_n \in S_c^n} \Pr_{\mathcal{M}^{n-1}}(\Diamond s_n) \cdot \left(\mathbb{E}_{\mathcal{M}^{n-1}, s_{n-1}}(\diamondplus\, s_n \mid \Diamond s_n) + \mathfrak{e}_{s_n}^{(n)}\right)\Big)\Bigg)\\
=& \mathfrak{e}_{s_0}^{(0)}\\
&+ \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \mathbb{E}_{\mathcal{M}^0}(\diamondplus\, s_1 \mid \Diamond s_1)\\
&\quad + \sum_{s_1 \in S_c^1} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \mathfrak{e}_{s_1}^{(1)}\\
&+ \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Pr_{\mathcal{M}^1}(\Diamond s_2) \cdot \mathbb{E}_{\mathcal{M}^1, s_1}(\diamondplus\, s_2 \mid \Diamond s_2)\\
&\quad + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \Pr_{\mathcal{M}^1}(\Diamond s_2) \cdot \mathfrak{e}_{s_2}^{(2)}\\
&\dots\\
&+ \sum_{s_1 \in S_c^1} \dots \sum_{s_n \in S_c^n} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \ldots \cdot \Pr_{\mathcal{M}^{n-1}}(\Diamond s_n) \cdot \mathbb{E}_{\mathcal{M}^{n-1}, s_{n-1}}(\diamondplus\, s_n \mid \Diamond s_n)\\
&\quad + \sum_{s_1 \in S_c^1} \dots \sum_{s_n \in S_c^n} \Pr_{\mathcal{M}^0}(\Diamond s_1) \cdot \ldots \cdot \Pr_{\mathcal{M}^{n-1}}(\Diamond s_n) \cdot \mathfrak{e}_{s_n}^{(n)}\\
=& \mathfrak{e}_{s_0}^{(0)}
\end{aligned}$$

$$
\begin{aligned}
&+ \sum_{s_1 \in S_c^1} \mathscr{E}^{(0)}_{s_0,s_1} + \sum_{s_1 \in S_c^1} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \mathfrak{e}^{(1)}_{s_1} \\
&+ \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \mathscr{E}^{(1)}_{s_1,s_2} + \sum_{s_1 \in S_c^1} \sum_{s_2 \in S_c^2} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \mathscr{P}^{(1)}_{s_1,s_2} \cdot \mathfrak{e}^{(2)}_{s_2} \\
&\dots \\
&+ \sum_{s_1 \in S_c^1} \dots \sum_{s_n \in S_c^n} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \dots \cdot \mathscr{P}^{(n-2)}_{s_{n-2},s_{n-1}} \cdot \mathscr{E}^{(n-1)}_{s_{n-1},s_n} \\
&\quad + \sum_{s_1 \in S_c^1} \dots \sum_{s_n \in S_c^n} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \dots \cdot \mathscr{P}^{(n-2)}_{s_{n-2},s_{n-1}} \cdot \mathscr{P}^{(n-1)}_{s_{n-1},s_n} \cdot \mathfrak{e}^{(n)}_{s_n} \\
=&\mathfrak{e}^{(0)}_{s_0} \\
&+ \sum_{s_1 \in S_c^1} \mathscr{E}^{(0)}_{s_0,s_1} + \sum_{s_1 \in S_c^1} \mathscr{P}^{(0)}_{s_0,s_1} \cdot \mathfrak{e}^{(1)}_{s_1} \\
&+ \sum_{s_2 \in S_c^2} \left(\mathscr{P}^{(0)} \cdot \mathscr{E}^{(1)}\right)_{s_0,s_2} + \sum_{s_2 \in S_c^2} \left(\mathscr{P}^{(0)} \cdot \mathscr{P}^{(1)}\right)_{s_0,s_2} \cdot \mathfrak{e}^{(2)}_{s_2} \\
&\dots \\
&+ \sum_{s_n \in S_c^n} \left(\mathscr{P}^{(0)} \cdot \dots \cdot \mathscr{P}^{(n-2)} \cdot \mathscr{E}^{(n-1)}\right)_{s_0,s_n} \\
&\quad + \sum_{s_n \in S_c^n} \left(\mathscr{P}^{(0)} \cdot \dots \cdot \mathscr{P}^{(n-1)}\right)_{s_0,s_n} \cdot \mathfrak{e}^{(n)}_{s_n} \\
=&\sum_{i=1}^{n} \sum_{s \in S_c^i} \left(\prod_{j=0}^{i-2} \mathscr{P}^{(j)} \cdot \mathscr{E}^{(i-1)}\right)_s + \sum_{i=0}^{n} \sum_{s \in S_c^i} \left(\prod_{j=0}^{i-1} \mathscr{P}^{(j)}\right)_s \cdot \mathfrak{e}^{(i)}_s \\
=&\sum_{i=1}^{n} \sum_{s \in S_c^i} \left(\prod_{j=0}^{i-2} \mathscr{P}^{(j)} \cdot \mathscr{E}^{(i-1)}\right)_s + \sum_{i=0}^{n} \left(\prod_{j=0}^{i-1} \mathscr{P}^{(j)} \cdot \mathfrak{e}^{(i)}\right).
\end{aligned}
$$

□

Corollary 4.15. *If for all $s \in \xi$: $s(c) = n$, then for all $i < n$: $\mathfrak{e}^{(i)} = \mathfrak{p}^{(i)} = \vec{0}$. Thus,*

$$
\Pr_{\mathcal{M}}(\Diamond \xi) = \prod_{j=0}^{n-1} \mathscr{P}^{(j)} \cdot \mathfrak{p}^{(n)}
$$

and if $\Pr_{\mathcal{M}}(\Diamond \xi) = 1$

$$
\mathbb{E}_{\mathcal{M}}(\Diamond\!\!\!\!\!\!+\,\xi) = \sum_{i=1}^{n} \sum_{s \in S_c^i} \left(\prod_{j=0}^{i-2} \mathscr{P}^{(j)} \cdot \mathscr{E}^{(i-1)}\right)_s + \left(\prod_{j=0}^{n-1} \mathscr{P}^{(j)} \cdot \mathfrak{e}^{(n)}\right).
$$

Factorization for Simple, Observing Counters

A simple counter is observing, if the counter value has no effect on the structure, probabilities, and rewards of the model, except for the minimal or maximal counter value.

Definition 4.16 (Observing Counters). *A simple counter c of $\mathcal{M}$ with maximal value n is observing, if for each $0 < i, j < n$ the factors $\mathcal{M}^i$ and $\mathcal{M}^j$ are bisimilar with respect to the labeling $\mathsf{L}(s) = s{\downarrow}_{\mathit{Vars}\setminus\{c\}}$, in the sense of [Seg95].*

Thus, for a simple, observing counter all factors except for the first and the last one are bisimilar with respect to the labeling that ignores the counter values.

Corollary 4.17. *Let c be a simple, observing counter of $\mathcal{M}$ with update value 1. For all $s \in \xi$, let $s(c) = n$. Let $S_c^e = S_c^1$, $\mathscr{P} = \mathscr{P}^{(1)}$ and $\mathscr{E} = \mathscr{E}^{(1)}$. Then:*

$$\Pr_{\mathcal{M}}(\Diamond\xi) = \mathscr{P}^{(0)} \cdot \mathscr{P}^{n-1} \cdot \mathfrak{p}^{(n)}$$

and if $\Pr_{\mathcal{M}}(\Diamond\xi) = 1$

$$\mathbb{E}_{\mathcal{M}}(\diamondplus\xi) = \sum_{s \in S_c^e} \mathscr{E}_s^{(0)} + \sum_{i=2}^{n} \sum_{s \in S_c^e} \left(\mathscr{P}^{(0)} \cdot \mathscr{P}^{i-2} \cdot \mathscr{E}\right)_s + \left(\mathscr{P}^{(0)} \cdot \mathscr{P}^{n-1} \cdot \mathfrak{e}^{(n)}\right)$$

Proof. Since c is a simple, observing counter with update value 1, for all $0 < i, j < n$ we have that $\mathcal{M}^i$ and $\mathcal{M}^j$ are bisimilar with respect to the labeling that erases the counter value, and thus $\mathscr{P}^{(i)} = \mathscr{P}^{(j)} = \mathscr{P}$ and $\mathscr{E}^{(i)} = \mathscr{E}^{(j)} = \mathscr{E}$. This yields the corollary. □

To compute $\mathscr{P}^k$, $k-1$ matrix multiplications would be necessary with a naive algorithm — which would cost a lot of time for large k. The well-known square-and-multiply algorithm [Knu11] calculates the power with a logarithmic number of matrix multiplications.

When computing $\mathbb{E}_{\mathcal{M}}(\diamondplus\xi)$ the first summand can be computed fast, and the third summand is calculated using the square-and multiply algorithm. For the second summand, we first observe that

$$\begin{aligned}
&\sum_{i=2}^{n} \sum_{s \in S_c^e} \left(\mathscr{P}^{(0)} \cdot \mathscr{P}^{i-2} \cdot \mathscr{E}\right)_s \\
&= \sum_{s \in S_c^e} \left(\mathscr{P}^{(0)} \cdot \sum_{i=2}^{n} \mathscr{P}^{i-2} \cdot \mathscr{E}\right)_s \\
&= \sum_{s \in S_c^e} \left(\mathscr{P}^{(0)} \cdot \sum_{i=0}^{n-2} \mathscr{P}^{i} \cdot \mathscr{E}\right)_s \\
&= \sum_{s \in S_c^e} \left(\mathscr{P}^{(0)} \cdot \left(\mathscr{E} + \sum_{i=1}^{n-2} \mathscr{P}^{i} \cdot \mathscr{E}\right)\right)_s.
\end{aligned}$$

We carry over the idea of square-and-multiply to compute $\sum_{i=1}^{n-2} \mathscr{P}^i$ with a logarithmic number of matrix multiplications. Observe, that for an arbitrary matrix A and natural numbers $k, k_1, k_2 \in \mathbb{N}$ such that $k = k_1 + k_2$ it holds:

$$\sum_{i=1}^{k} A^i = \sum_{i=1}^{k_1} A^i + A^{k_1} \sum_{i=1}^{k_2} A^i. \tag{DS}$$

Thus, analogously to square-and-multiply, $\sum_{i=1}^{k} A^i$ can be composed using the binary representation of $k = \sum_{j=0}^{m} a_j \cdot 2^j$ for unique $a_0, \ldots, a_m \in \{0, 1\}$. We use an index set $I = \{j \mid a_j = 1\}$, and set $k_1 = 2^j$ such that j is the smallest index in I and $k_2 = \sum_{j' \in I, j' > j} 2^j$. Then, the second summand can be composed recursively in an analogous way. Thus, computation of the complex sum can be reduced to calculating and summing up a series of values $\sum_{i=1}^{k'} A^i$ for $k' \in \{2^j \mid 0 \leq j \leq m\}$. Each of these sums can be calculated re-using values of already computed sums with smaller index.

Example 4.18. *Consider an arbitrary matrix A and $k = 23 = 16 + 4 + 2 + 1$:*

$$\sum_{i=1}^{23} A^i = \underbrace{(A^1 + \ldots + A^{16})}_{\sum_{i=1}^{16} A^i} + \Big(\underbrace{(A^1 + A^2 + A^3 + A^4)}_{\sum_{i=1}^{4} A^i} + \big(\underbrace{(A + A^2)}_{\sum_{i=1}^{2} A^i} + \underbrace{(A)}_{\sum_{i=1}^{1} A^i} \cdot A^2\big) \cdot A^4\Big) \cdot A^{16}.$$

$A + A^2$ can be computed as $A + A \cdot A$. Analogously, $A + \ldots + A^4$ can be computed reusing $z = A + A^2$, which yields $A + \ldots + A^4 = z + z \cdot A^2$.

At the beginning of this chapter we outlined that the factorization approach will be applied to the redo-based fault-tolerance protocol presented in the next chapter. The counter of this PPG is a simple, observing counter with update value 1. Thus, Corollary 4.17 and the computation simplification for $\sum_{i=1}^{n-2} \mathscr{P}^i$ yield the mathematical basis to apply factorization and opens the way for resiliency configuration of the redo-based fault-tolerance protocol.

4.4 Implementation

One purpose of this work is to provide a tool that implements the factorization approach as it is needed for the configuration presented in the next chapter. The implementation shall not only support models for redo-based fault tolerance, but is meant to be the starting point of counter-based factorization of arbitrary PPGs with counters. This purpose resulted in `fact`, a tool for counter-based factorization written in python. `fact` is a tool that, given a purely probabilistic program graph and a probabilistic reachability property or expected accumulated reward, searches for a simple, observing counter in the model and applies the factorization approach presented before.

Overview. `fact` is a object-oriented tool that takes as input a (parametric) probabilistic program graph in the common PRISM language [11b], together with a probabilistic reachability property or an expected accumulated reward, also in PRISM language. The current implementation of `fact` makes use of the probabilistic model checker Storm [Deh+17] (written in C++) and the python-based computer algebra system (CAS) sympy [Meu+17]. `fact` implements interfaces for Storm and sympy. Thus, `fact` can be easily extended to support other model checker or computer algebra systems by defining new interfaces. The output of `fact` is a rational function, or, if the model does not have free parameters, a result value for the input property. The input, output, interfaces and the work flow of `fact` is visualized in Figure 4.7.

An important characteristic of `fact` is that almost all necessary steps are performed on the PPG, without building the induced Markov chain. Some steps need to be performed on the factors of the induced Markov chain, but due to the observability of the counter and the resulting bisimilarity of the factors, it is not necessary to explore the complete state space of the DTMC.

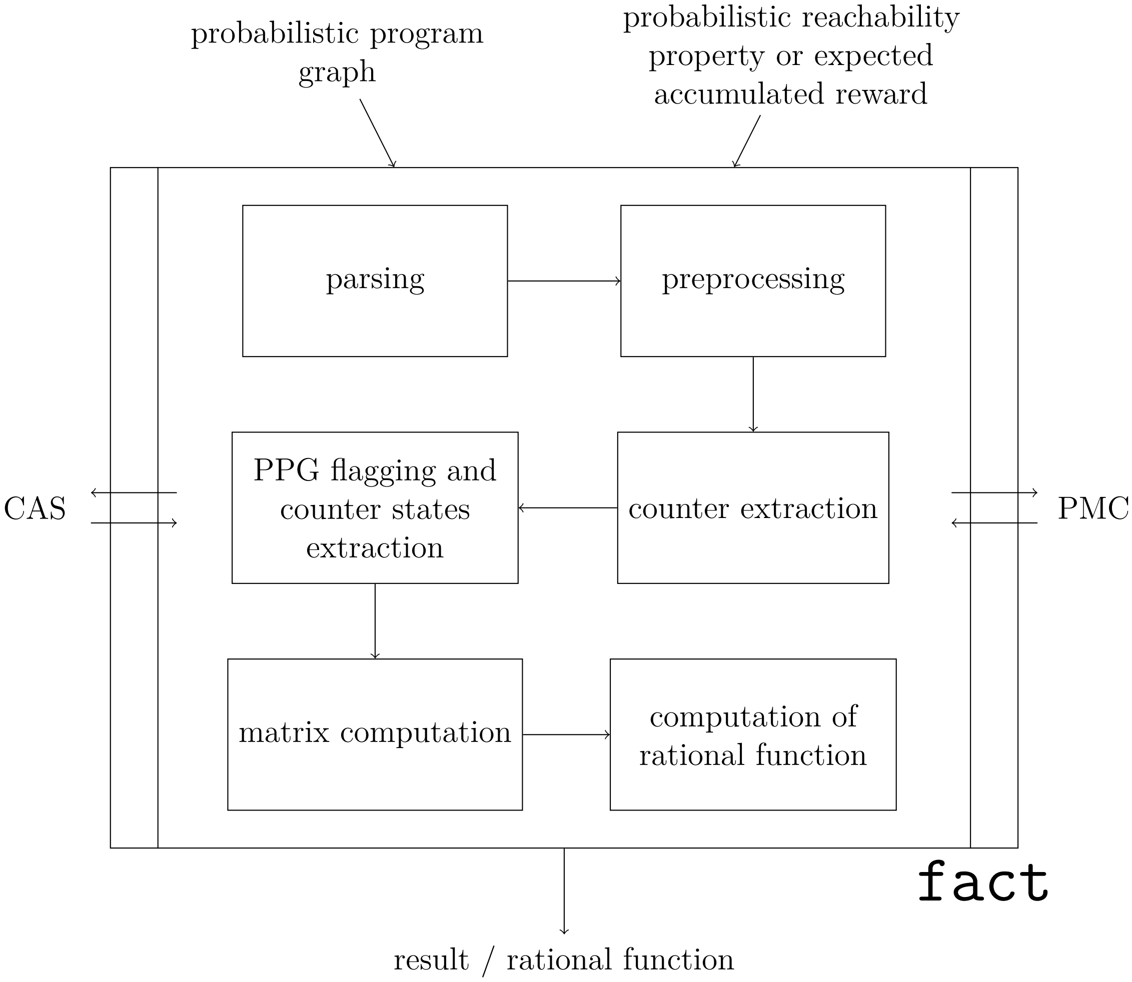
probabilistic program graph
probabilistic reachability property or expected accumulated reward
parsing
preprocessing
CAS
PPG flagging and counter states extraction
counter extraction
PMC
matrix computation
computation of rational function
fact
result / rational function

Figure 4.7: Input, output, interfaces and work flow of `fact`.

The work flow of `fact` starts with parsing the model and the property. The PPG is parsed, and it is checked whether the property is a reachability property. After parsing, some factorization-independent preprocessing steps, that ease the application of PMC to parametric models, are performed. Then, the variables of the PPG are analyzed to find simple, observing counters. A counter is chosen, and the PPG is flagged to identify counter states in the induced DTMC. These states are found by exploring a small part of the DTMC's state space and stored for later use. The PPG is factorized, and local characteristics of the factors are computed via Storm. Results are extracted from the Storm-output and collected in matrices. Finally, the global rational function is computed via the CAS sympy from the local characteristics.

The Prism Language. The Prism language enables easy, compositional description of PPGs. A PPG in Prism language consists of one ore more *components*, each describing a single PPG. For each component, a finite set of variables and their domain are defined. The domain can be a subset of successive integers (defined by a lower and an upper bound), or $\{\mathit{true}, \mathit{false}\}$. These variables correspond to the local variables LV from the definition of PPGs in Section 2. Furthermore, a component consists of a set of *transitions*, each consisting of an action name, a guard (a propositional formula), and a probabilistic update on the local variables. A probabilistic update is a list of probability - update pairs, where each update is a set of variable updates of the form $x' = f(\mathfrak{e})$. This update denotes that in the induced DTMC, when being in state $\mathfrak{e}$, the value of the variable x in the next state is defined as $f(\mathfrak{e})$. Variable updates of the form $x' = x$ are omitted.

The Prism language supports the use of *constants*. A constant can be seen as a variable with single-elemented domain, i.e., no PPG component can change a constants value. Transition probabilities, updates, and rewards can be specified in terms of functions that depend on constant values. Constants that are used only in transition probabilities and rewards can be left undefined and will be parameters during PMC.

Reward structures in the PRISM languages map sets of variable evaluations to rational values. The sets of variable evaluations are described via propositional formulas. This mapping does not have to be total. For each evaluation, i.e., for each state in the induced DTMC, the reward of this evaluation is the sum of all rational numbers assigned to propositional formulas that are satisfied by this evaluation.

For more details on the Prism language, see [11b].

Parsing. We implemented an LR(1)-parser to parse a PPG written in the Prism language. The parser generation was simplified by using of the python parser library lark [18b]. The parser is capable of parsing both the input model and the input property.

The parser supports almost the full Prism language, yet some parts, that are not relevant for this work, are omitted. For example, the Prism language allows for defining more advanced forms of compositions, that are out of context of this work and thus are not supported.

We assume that the parsed PPG is purely probabilistic, which can be easily checked by inspecting whether the PPG description is preceded by the keyword **dtmc**. A non-deterministic program graph whose description is preceded by **dtmc** will be transformed in a purely probabilistic program graph by Storm, by replacing all nondeterminism with equally distributed probabilistic choices. The parser keeps the structure of the Prism model, including, e.g., variable ordering.

The parser is able to parse the full range of LTL expressions, yet we check whether the input property is a reachability property, by testing whether the LTL-formula is of the form $\Diamond\psi$ and whether ψ is a propositional formula.

For all variables that will be handled as counters during factorization, we assume that their domain is a range of integers, and that the lower bound is 0.

The result of both, model parsing and property parsing, is an abstract syntax tree (AST) of python objects. `fact` supports arbitrary precision and exact representation of rational numbers by parsing all rational numbers as strings and converting them to sympy objects in the respective precision directly. The precision can be controlled via command line and describes the number of digits that shall be preserved. Thus, the argument of the command-line switch `--precision` is an arbitrary natural number, or "exact".

Preprocessing. Transition probabilities or rewards can be defined as arithmetic expressions. Some of these expressions are not supported by Storm, or cause long run-times when using Storm for parametric models. When replacing these expressions with constants, and treating these constants as parameters, a significant speed up can be obtained. `fact` supports automated extraction of such problematic transition probabilities or reward definitions, and replaces them with undefined constants, such that they are handled as parameters during PMC. `fact` traverses the AST of the model and for each mathematical expression in the transition probabilities or rewards that satisfies one of the conditions below, it inserts a new, undefined constant, and replaces the mathematical expression by this new constant's name. The conditions for being handled parametric are:

- The expression is a power with an exponent larger than some natural number m, which can be specified using `--parametrize-exponents-above` "m".
- The expression is a power with undefined exponent, i.e., the exponent is a function containing some undefined constant. (This is not supported by Storm.)
- The expression is a power with a non-integer exponent. (This is not supported by Storm.)
- The expression is the definition of some constant that is in the argument list of `--parameters`.

Counter Extraction. The goal of this step is to find all variables that can be handled as counters. `fact` currently supports only the extraction of simple counters with update value 1. A variable c is a simple counter, if for each transition that updates the counter, the update is of the form $c' = c + 1$. The counter is observing, if for each transition the transition probabilities do not depend on the counter value, and if the guard depends on the counter value, then all occurrences of the counter variable are of the form $v = n$ or $v < n$, where n is the maximal value of the counter.

`fact` supports factorization only for one counter. In case of several simple observing counters with update value 1, we choose the one with largest maximal value. Since we assume the minimal value to be 0, this counter induces the largest number of factors in the DTMC, and thus heuristically the most significant reduce in the size of the models where PMC is applied.

Counter Flagging. In the next step, we insert a Boolean variable f, that tracks whether the counter c has been updated. Let LV be the set of local variables of the component that defines the counter. We define a new variable f with domain $\{\mathit{true}, \mathit{false}\}$ and initial value *false*, and add it to the set of local variables. For every transition in this component, and for each update u that increments the counter value, we transform the update such that it assigns f the value *true*, i.e., we add the variable update $f' = \mathit{true}$. For each update that does not change the value of c, we transform the update such that it assigns f the value *false*. Furthermore, for each action α which is not in the set of actions of this component, we add a new transition t to the counter-defining component, with $t = \mathit{true} \overset{\alpha}{\hookrightarrow} \mathit{dirac}_{\mathrm{u}}$ for the update $\mathfrak{u} \in \mathit{Upd}(LV)$ that assigns f the value *false* and does not change the value of other variables.

Counter States. To apply factorization, we identify all counter states in S_c^e and store them as State-objects. A state in `fact` is a mapping (a python dictionary), assigning each variable of the PPG a value. To find all counter states, we explore a part of the induced DTMC's state space.

The procedure `build_model_partially(`$\mathcal{P}$`,condition,S)` (see Algorithm 1) takes as input a PPG $\mathcal{P}$, a propositional formula `condition`, and a set of states `S`. Starting from the states in `S`, it explores the state space (i.e., determines the set of reachable states), but treats states that do not satisfy `condition` as absorbing states. The procedure maintains sets of states `dont_explore` and `explored`, which are initially empty. `dont_explore` will be used to store states that do not satisfy the `condition`. Furthermore, there is a set of states `unexplored`, which is initially the input set `S`. As long as there is a state in the list of unexplored states, this state is removed from this list. If the state does not match the condition, it is added to `dont_explore`. Otherwise, it is added to `explored` and is explored: All transitions of the PPG are traversed, and for each transition where the guard is satisfied by the state, all updates in the probabilistic update with probability greater than 0 are applied to the state. Thus, a new set of states is obtained. Each state in the new set of states is added to the set of unexplored states, if it is not yet in one of the set `dont_explore` or `explored`. When the set of unexplored states is empty, the procedure `build_model_partially(`$\mathcal{P}$`,condition,S)` returns the state set `dont_explore`.

Lemma 4.17 requires all factors but the first and the last one to be bisimilar with respect to the labeling arising from the variable evaluations, but ignoring the counter value. In the implementation we allow parts of the bisimilar factors to be not reachable (cf. Figure 4.8). The set S_c^e is now defined as the set of states $\{s{\downarrow_{\mathit{Vars}\setminus\{c\}}} \mid s \in S_c\}$, i.e., the set of counter-state representations that arise from reachable counter states, when ignoring the counter value. If we would explore the reachable state space of the first factor $\mathcal{M}^0$, and set $S_c^e = S_c^1$, counter state representations might be missing, since some counter states might not be reachable in the first factor.

Algorithm 2 is used to find all counter state representations, i.e., the set S_c^e. It initializes a set `counter_states` as the empty set, and calls `build_model_partially` with the PPG, the condition that the counter value (of some state) is smaller than 1, and the singleton state set containing the initial state. The states in the returned list `dont_explore` are counter states with value 1. We store these counter states in `counter_states`. Yet, it might be that some counter states in S_c^e are not reachable from the initial state directly, i.e., with only one counter update. Thus, these states are not included in the list `dont_explore` returned by `build_model_partially`.

We again call `build_model_partially`, now with the condition that the counter value is smaller than 2, and the previously retrieved set of counter states. The procedure will now explore all counter states that where not explored in the last call, hence it will build factor $\mathcal{M}^1$. We obtain a new list of counter states, all having counter value 2. For each of this new states, we check whether a bisimilar state is already contained in `counter_states`. If so, we ignore it, if not we add the new state to `counter_states` and to a set of states `explorestates`. If no new state was added to `counter_states`, all states in S_c^e are found and we return `counter_states`. Otherwise, `build_model_partially` is called again, with

Algorithm 1: The procedure build_model_partially($\mathcal{P}$,condition,unexplored) explores the state space of the induced DTMC of $\mathcal{P}$ from the states in unexplored, but treats states that do not satisfy the condition as absorbing states.

```
input  : A PPG $\mathcal{P}$, a propositional formula condition, a list of states
         unexplored
output : a list of unexplored states
dont_explore:= $\emptyset$;
explored:= $\emptyset$;
while unexplored $\neq \emptyset$ do
    s:= unexplored.pop();
    if s $\not\models$ condition then
        dont_explore.add($s$);
    else
        explored.add($s$);
        new_states:= s.explore();
        for s' $\in$ new_states do
            if s' $\notin$ explored and s' $\notin$ dont_explore then
                unexplored.add(s');
return (dont_explore);
```

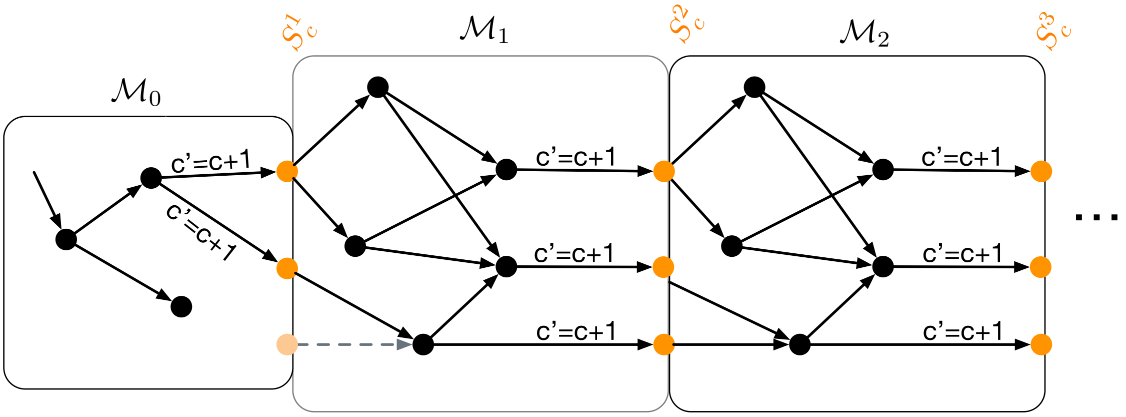

Figure 4.8: Factors $\mathcal{M}^0$, $\mathcal{M}^1$, and $\mathcal{M}^2$ arising from a simple, observing counter c. Counter states are marked orange. In the first factor, two counter states (states in S_c^1) are reachable. For these states there are bisimilar states in $S_c^2, S_c^3, \ldots$ In S_c^2 there is a counter state that is reachable in factor $\mathcal{M}^1$, but there is no *reachable* bisimilar state in S_c^1. Nevertheless, there is a *non-reachable* bisimilar state.

the condition that the counter value is smaller than 3, and the set `explorestates` of new counter states that were previously added to `counter_states`. A new set of counter states is retrieved, and so on. The while loop is left as soon as all new counter states are already represented in `counter_states`. Finally, for all counter states, the value of the counter is set to 1 to obtain counter state representations with equivalent counter value, and the set of counter state representations is returned.

Algorithm 2: Algorithm extract_counter_states($\mathcal{P}$,c) retrieves all counter state representations in a PPG for a simple, observing counter with update value 1.

```
input  : A PPG $\mathcal{P}$, a counter $c$
output : A list of counter states $S_c^e$
counter_states:= $\emptyset$ explore_states:= {$\mathcal{P}$.initial_state };
i := 1;
while explore_states $\neq \emptyset$ do
    condition = $s(c) < i$;
    new_states:= build_model_partially ($\mathcal{P}$,condition,explore_states);
    explore_states:= $\emptyset$;
    for s $\in$ new_states do
        if not is_represented (s,counter_states) then
            counter_states.add(s);
            explore_states.add(s);
    i := i+1;
for s $\in$ counter_states do
    s (c):=1;
return (counter_states);
```

Computation of Factor Matrices. To compute the matrix entries, Storm is called, taking as input a PPG representing a factor of the original PPG, and the property that shall be computed. The factorization approach for simple, observing counters with update value 1 requires the computation of the matrices $\mathscr{P}$ and $\mathscr{E}$, and the vectors $\mathscr{P}^{(0)}$, $\mathscr{E}^{(0)}$, $\mathfrak{p}^{(n)}$ and $\mathfrak{e}^{(n)}$.

To retrieve the PPG representing the first factor $\mathcal{M}^0$, we transform $\mathcal{P}$ such that all counter states are absorbing. For this purpose, we iterate all transitions and for each guard g, we modify the guard to be $g \wedge (c = 0)$. Hence, transition guards in the transformed PPG are only satisfiable by states s with $s(c) = 0$. Since for all counter states s_c we have $s_c(c) \geq 1$, all counter states are absorbing, and the transformed PPG represents the factor $\mathcal{M}^0$ and can be utilized to compute $\mathscr{P}^{(0)}$, and $\mathscr{E}^{(0)}$.

We set up two files, one containing the set of properties $\{\Pr_{\mathcal{M}}(\Diamond s) \mid s \in S_c^e\}$ and one containing $\{\mathbb{E}(_{\mathcal{M}}\oplus s \mid \Diamond s) \mid s \in S_c^e\}$. To compute the values of $\mathscr{P}^{(0)}$ and $\mathscr{E}^{(0)}$, we call Storm on the transformed PPG (and the original initial evaluation) and the respective properties file. The results, a series of rational functions, is extracted from the Storm log and stored as text in two lists. One of these lists represents $\mathscr{P}^{(0)}$, and the elementwise product of both lists represents $\mathscr{E}^{(0)}$. Note that $\mathscr{P}^{(0)}$ and $\mathscr{E}^{(0)}$ are the same matrices as defined in Lemma 4.17, when assuming that $\mathcal{M}^i$ and $\mathcal{M}^j$ are bisimilar for all $1 \leq i \leq n$. If there are counter states in S_c^e that are not reachable in the first factor, $\mathscr{P}^{(0)}$ and $\mathscr{E}^{(0)}$ have additional lines containing zeros, since the probability of reaching these states and the conditional expected reward is 0 for these states.

For the computations of $\mathscr{P}$ and $\mathscr{E}$, we use factor $\mathcal{M}^1$. To retrieve $\mathcal{M}^1$, we first change the previously transformed transition guards to $g \wedge c = 1$. Thus, states in S_c^2 are absorbing states. Iterating over all counter states $s_c \in S_c^e$, we set s_c as initial evaluation of the transformed PPG, and apply PMC. Since c is observing, and $s_c(c) = 1$ was set in Algorithm 2, the PPG with initial evaluation s_c now represents factor $\mathcal{M}^1$. If s_c is not reachable in $\mathcal{M}^1$, we can use this factor to compute the probability and expected accumulated rewards anyway. The observability of the counter guarantees the results to be correct. When setting s_c as initial state, $\mathcal{M}^1$ will have the same properties as all $\mathcal{M}^i$ where the respective bisimilar state in S_c^i is reachable.

Storm is called for each s_c on the transformed PPG and the probabilistic properties input file (with adjusted counter values), and results are extracted as before, now stored in a list of lists that represents the matrix. The computation of $\mathscr{E}$ is analogous.

Finally, we to compute $\mathfrak{p}^{(n)}$ and $\mathfrak{e}^{(n)}$. We revert the previously performed transition transformation, i.e., for each transition's guard $g \wedge (c = 1)$, we transform the guard to be g. Furthermore, we set up two files, one containing the property $\Pr_{\mathcal{M}}(\Diamond \xi)$, and one containing $\mathbb{E}_{\mathcal{M}}(\diamondplus \xi)$. Then, for each counter state $s_c \in S_c^e$, we transform s_c such that $s_c(c) = n$ and set s_c as initial evaluation of the PPG. Thus, the PPG now represents the factor $\mathcal{M}^n$ with initial state $s_c \in S_c^n$. For each s_c and each of the two files we call Storm on the PPG with initial value s_c and the respective file. Thus, we obtain two lists, representing $\mathfrak{p}^{(n)}$ and $\mathfrak{e}^{(n)}$.

The Global Rational Function. After computing the vectors and matrices (which are still represented as lists, with text entries) that are necessary for factorization, the rational functions in these lists need to be transformed to sympy mathematical objects. This transformation is completely provided by sympy, including the construction of matrix objects from the lists. Yet, this transformation is rather time- and memory intensive, especially when the rational functions are large. Recall that `fact` automatically replaces transition probabilities by parameters, if they are not manageable by Storm or cause time issues. These free parameters may cause the rational functions to be larger than they would be without these additional parameters. During configuration of the redo-based fault-tolerance protocol, we observed that the time for transforming rational functions represented as text can be drastically reduced when, before applying this transformation, the free parameters that were introduced by `fact`, are replaced by their definitions. This replacement is performed on a textual level. We sort the name of the parameters such that parameters whose name is a sub-string of another parameter are replaced last. The behavior, when `fact` replaces parameters by their definition, can be controlled via the command-line switch `--replace-at` "v". "v" is one of the strings "after-PMC" (replace on a textual level), "after-mathparse" (replace after transforming the rational functions to mathematical objects), or "after-computation" (replace after computing the global characteristic).

After transforming the PMC-results to matrix objects that have rational functions as entries, the factorization formulas can be used to retrieve the global characteristic. To compute $\Pr(\Diamond\xi)$, sympy is utilized to retrieve the matrix power and the scalar products. To compute $\mathscr{P}^k$ with a logarithmic number of matrix multiplications, sympy implements the well-known square-and-multiply algorithm [Knu11].

Algorithm 3: Algorithm to compute $\sum_{i=1}^{k} A^i$

```
input  : a matrix A, a national number $k$
output : $\sum_{i=1}^{k} A^i$
if $k = 0$ then
    return 0;
i := $k - 1$;
n := 0;
z := $A$;
result:= $A$;
while $i \neq 0$ do
    if $i \mod 2 = 1$ then
        result:= $z +$ result $\cdot A^{2^n}$;
    z := $z + z \cdot A^{2^n}$;
    i := $\lfloor \frac{i}{2} \rfloor$;
    n := $n + 1$;
return result;
```

Algorithm 3 is a dynamic algorithm in the style of square-and-multiply to compute $\sum_{i=1}^{n-2} \mathscr{P}^i$ with a quadratic logarithmic number of multiplications. It structurally equals an iterative implementation of the square-and-multiply approach, but differs in the variable assignments. The main part of the algorithm is a while-loop, where in each iteration, a natural number n, which is initially 0, is increased by 1. It represents the exponents of the powers of two, as described in the previous section. A value i, initialized with k, is updated to $\lfloor \frac{i}{2} \rfloor$ each iteration. It is used to compute the values a_j: If $i \mod 2 = 1$, then $a_n = 1$ in the respective iteration. The variable z contains the values $\sum_{i=1}^{n} A^i$, result stores the intermediate result computed so far. In each iteration where $i \mod 2 = 1$, result is updated according to Equation (DS). Afterwards, z is updated (independently from $i \mod 2$) to represent the value $\sum_{i=1}^{2^{n+1}} A^i$, which will be used in the next iteration to update result and z.

With this implementation, we are now ready to apply the factorization approach to configure redo-based fault-tolerance protocols, as we do exemplarily in the next chapter.

5 Configuration of Redo-Based Fault-Tolerance Protocols

Every computation that is performed on a computer, every low-level operating service, and every high-level user application is, on an abstract level, a sequence of instructions that produces data. Many of these instruction sequences, in the further called *application*, and the produced data need to be reliable. If operating system services are not protected against bit flips, the system might become prone to malware, or data might be accessible from unauthorized users. For high level applications like bank services or airplane software, bit flips might cause dramatic financial loss or even harm human lives.

Redo-based fault-tolerance techniques provide protection of instruction sequences and of the produced data against bit flips. For this purpose, redundance is added to the instruction sequence before it is executed. The instruction sequence is, also before execution, split into several *transactions*. Then, during execution, each transaction is checked for bit-flip-caused errors in the instruction flow and in the produced data. If an error is detected, the transaction is redone.

This opens a configuration problem: How much redundance shall be added, such that the overhead arising due to maintaining the redundance is small, but a certain level of resilience can be provided? How many redos shall be performed? When a transaction is redone because of a detected error, this redo might again be affected by bit flips, another error might be detected and cause another redo. Thus, after how many of these successive, failed redos should the execution be aborted? And how many instructions shall be encapsulated in one transaction?

The goal of this chapter is to show how this configuration problem can be solved with probabilistic model checking. We exemplarily configure the former mentioned system variables of a redo-based fault-tolerance mechanism that is inspired by HAFT [Kuv+16], i.e., each instruction is a CPU-instruction. Some of these instructions write data to memory, and hence produce data. Redundance is added by duplicating a certain amount of instructions and a certain amount of data. The instruction duplicates enable detection of errors in the instruction flow, the data duplicates are used to find errors in the data.

Outline. The first section of this chapter gives more details about the principle of redo-based fault tolerance. In Section 5.2, we present a model family for redo-based fault-tolerance techniques. By setting model attributes, a model for a concrete redo-based fault-tolerance technique can be obtained. Finally, in Section 5.3, we present the exemplary configuration of the chosen redo-based fault-tolerance instance.

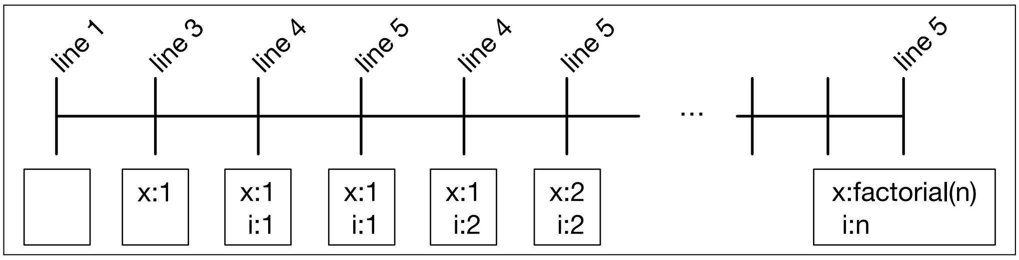

Figure 5.1: Control- and data flow of factorial(n).

5.1 The Concept of Redo-Based Fault Tolerance

An application is finite sequence of instructions, the *control flow*, that produces a sequence of data, the *data flow*. We distinguish control-flow instructions and data-flow instructions. Data-flow instructions are instructions that produce data (at the moment of execution), while control-flow instructions do not produce data. Data-flow instructions typically do also affect the control flow.

Algorithm 4: factorial(n): Factorial of a natural number n

```
  input  : A natural number $n$
  output : The factorial of $n$
1 if $n = 0$ then
2     return 1;
3 x:= 1;
4 for $i \in (1, \ldots, n)$ do
5     x:=x ·i;
6 return x;
```

Example 5.1 (An application computing the factorial). *Assume an application that computes the factorial of a natural number with Algorithm 4. Figure 5.1 shows the control flow (above the black line in the middle) and the data flow (below the black line) of this application. The data flow describes the changes in the memory of the algorithm. The if-clause and the return statement are control-flow instructions, since they do not produce data. In lines 3 and 5, a value is stored in variable x, thus these instructions are data-flow instruction. The head of the for-loop assigns values to variable i. Thus, this instruction is a data-flow instruction, too.*

Both the control flow and the data flow can be affected by errors. Errors, if not corrected, can cause the application to produce wrong data. Errors in the data flow obviously cause wrong data directly, while errors in the control flow have a high chance of affecting the data flow eventually (e.g., a wrong jump will very likely cause wrong data-flow instructions to be performed and thus cause errors in the data flow).

The primary goal of a redo-based fault-tolerance mechanism is to prevent an application from producing wrong output data. It enables error detection and correction in both the control-flow and the data flow. Redundance is added to detect errors. In this thesis, we consider instruction and data duplicates as redundance, i.e., instructions are duplicated and both the original instructions and the replicates are executed. The duplicated instructions operate on the duplicated data.

Error correction is performed by redoing parts of the application. To enable partial application redos, the application is partitioned into transactions before its execution. Transaction management instructions (e.g., begin-of-transaction, end-of-transaction) are included. In each transaction, original instructions and the data they produce are compared to their duplicates. If no error is detected in a transaction, the application's memory state is stored, and the next transaction is executed. If an error is detected in some transaction, the application's memory is reset to the state of the end of the previous transaction, and the faulty transaction is re-executed. During re-execution new errors may arise and cause further redos. A redo-based transaction mechanism typically comes with a pre-defined number of maximal redos that can be performed per transaction. If this number of redos does not suffice to correct the error, i.e., if there is an error in each redo, the application is aborted.

Example 5.2. *A redo-based fault-tolerance mechanism will partition the instruction flow into transactions. One possibility to do this is to encapsulate the if-clause in one transaction, the assignment* $x := 1$ *in another, and each iteration of the for-loop in a separate transaction. One way to provide resilience by adding redundance is, e.g., to duplicate each transaction and to run it twice, in parallel. At the end of each transaction the data produced by the original transaction and by the copied transaction is compared. If a mismatch is detected, both the original transaction and its duplicate are redone.*

The duplicates of instructions can also be affected by errors. Thus, transaction redos might be imposed although no error occurred in the original application. We say that the fault-tolerance mechanism has a *false positive* in this situation. If, on the other hand, an original instruction is erroneous, but this error can not be detected (e.g., if the same error occurs in the duplicate or if the instruction is not duplicated), the fault-tolerance mechanism has a *false negative*. If an error is not detected, and thus an erroneous transaction is not corrected, this error becomes a failure and can not be repaired by succeeding transactions. Ones the application has a failure, this failure will persist until the end of the application.

Error Model. Due to the nature of formal methods we do not need to impose restrictions on the type or number of errors occurring but simply assume the probability of an error affecting an instruction to be known. The presence of a failure might increase the probability of succeeding error occurrence. We also assume that this increased error probability is given.

Overhead. Error detection and correction causes overhead. This overhead can be measured, e.g., in terms of time or energy spend on error handling. The nature of formal methods allows to easily define several overhead measurements without changing the underlying model. Although the configuration in Section 5.3 focuses on time overhead, the same procedure can be performed for other overheads. Nevertheless, we need to identify the phases in the application run where overhead is produced. We only consider overhead that is produced during the application run, i.e., the overhead that is needed to insert redundance and transaction management instructions before execution is neglected. Overhead is generated by

- the transaction mechanism, i.e., by the execution of transaction management instructions,
- executing duplicated instruction,
- comparing original instructions and their duplicates (control-flow checking),
- comparing data produced by original instructions and duplicated instructions (data-flow checking),
- reloading the application's memory before transaction redo, and
- transaction redos.

Configuration. The main goal of a fault-tolerance technique is to protect an application against errors in the data-flow, i.e., to avoid false negatives, with an acceptable overhead. Redo-based fault-tolerance mechanisms as described above come with the following configuration parameters:

- The maximal number of redos per transaction: Performing more redos increases the chance of completing a transaction, but also increase the chance of having errors and false negatives, since more instructions are performed.
- The number of instructions in a transaction: Short transactions cause more overhead due to storing the application's memory after each transaction, but in long transactions there is a higher chance per transaction of having an error and thus a higher chance of failures and redos. Furthermore, redos are more costly when transactions are longer.
- The number of instructions to be duplicated: The more instructions are duplicated, the more overhead is generated, but false negatives are less likely.

5.2 A Model for Redo-Based Fault Tolerance

We give a detailed insight in the model we set up for analysis. This model contains adjustable *attributes*, which are summarized in Tables 5.9 and 5.10 at the end of this section. Assigning precise values to these attributes defines a concrete redo-based fault-tolerance mechanism, a concrete error model, and a concrete application. Probabilistic attributes and reward attributes can be left undefined, and then will be parameters when performing PMC on the model.

The formal model consists of components for the underlying hardware, the application, and the fault-tolerance protocol. The latter contains sub-modules for a control-flow checker (CFC), a data-flow checker (DFC), and a transaction redo manager (TRM) implementing the redo/abort-schema (cf. Figure 5.2).

Application model. An application performs a fixed number on instructions, which can be specified by the attribute *inst_num*. The instruction flow is separated into transactions, each consisting of *ta_len* instructions. These attributes define the number of transactions to be performed, *ta_num* = *inst_num* / *ta_len*. We distinguish three types of transactions. First, there are control-flow instructions, where errors affect only the control flow (e.g., jump

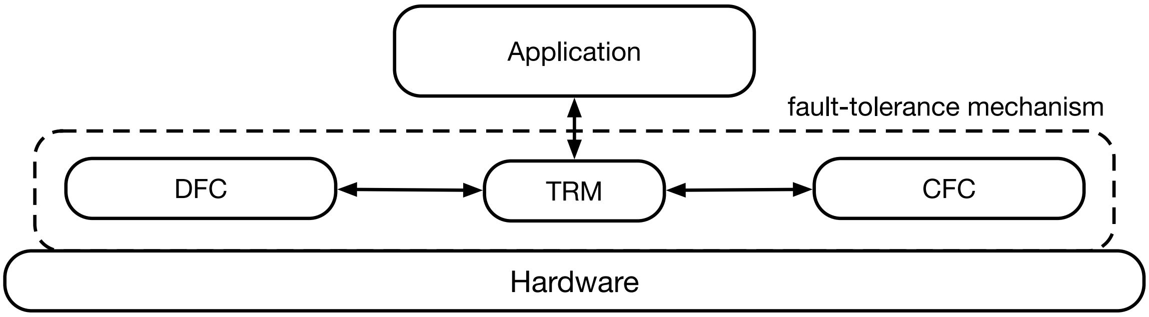

Figure 5.2: The structure of our model.

or return statements). Second, there are data-flow instructions, where errors affect both the control-flow and the data flow (e.g. add). Third, there are transaction management instructions, introduced to implement the transaction mechanism (e.g., begin-of-transaction and end-of-transaction). For the latter, errors do only affect the control flow. The ratio of control-flow and data-flow instructions can be set via the attribute *cf_df_ratio*. The amount of transaction management instructions per transaction is controlled by the attribute *tmi_num*.

Transaction execution is modeled as an atomic step, named PERFORM_TA (cf. Figure 5.3), in which the local counter variable *ta_counter* is increased. We use an ! to denote the sender of a synchronization action, while a ? is used on the receiver side. In case of PERFORM_TA, the receiver side is played by the hardware component (see paragraph "hardware model" below) that synchronizes on PERFORM_TA.

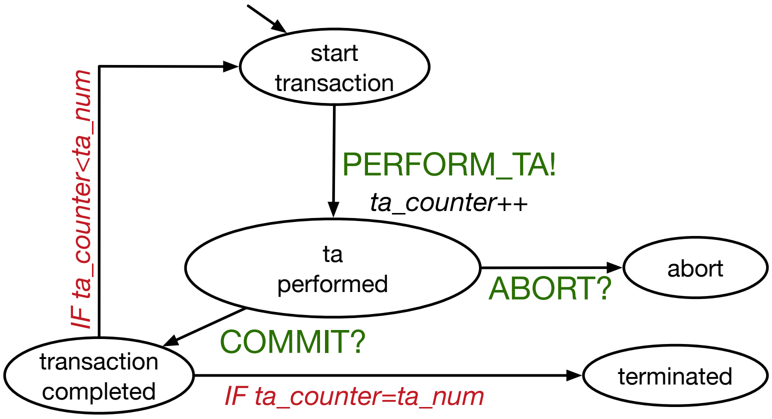

Figure 5.3: The program graph of the application.

After performing a transaction, the application is paused. During this pause, error detection and correction is performed by the TRM, DFC, and CFC. After this, the TRM sends either a COMMIT or an ABORT signal to the application, by executing the respective action. The application synchronizes on this action, either ending in an abort location or reaching location "transaction completed". In the latter case, if *ta_counter* did not yet reach *ta_num*, the application's location is changed to "start" and the next transaction is performed. Otherwise, location "terminated" is reached and the application stops.

TRM model. The transaction redo manager is the coordinative center of the fault-tolerance protocol. Its attribute *max_redos*, controlling the maximal number of redos, can be set to any natural number including zero, or ∞, indicating that infinitely many redos can be performed. After each transaction performed by the application, the transaction redo manager initiates checks on the data flow and the control flow, launches redos if necessary, and finally sends a COMMIT or ABORT signal to the application.

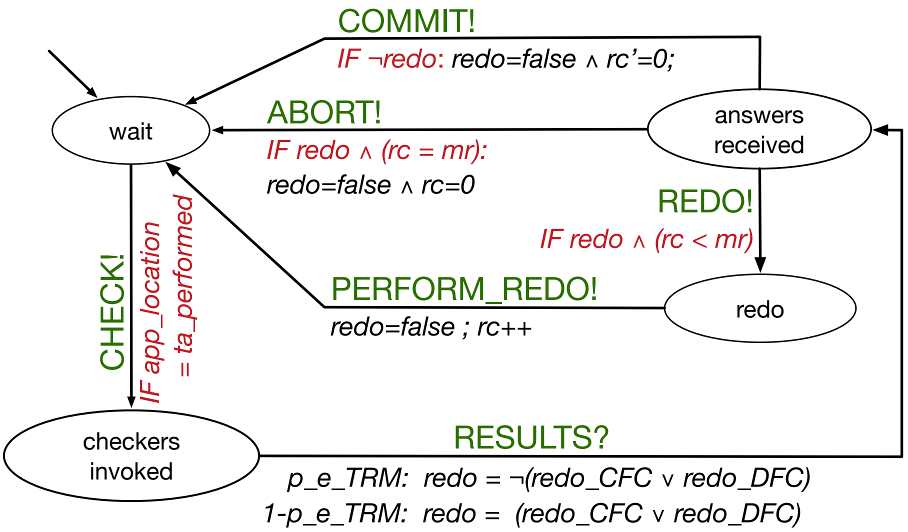

Figure 5.4: The program graph of the TRM. rc is short for *redo_counter*, mr abbreviates *max_redos*.

In our model, the TRM (cf. Figure 5.4) invokes the DFC and CFC with the synchronous action CHECK simultaneously as soon as the transaction has been performed. It then waits until within action RESULTS the results of both checkers are collected and evaluated in location "answer received". The TRM is also prone to errors, and thus might "misread" the signals received from the checkers.

The next action depends on whether an error was detected and whether an error in the TRM has occurred. If an error was detected by one of the checkers, and no error in the TRM, e.g., falsified the signal, a redo is initiated (if the maximal number of redos is not yet exceeded). The probability of an error in the TRM can be controlled by the attribute *p_e_TRM*. If either no error was detected, or an error was detected, but another error in the TRM occurred, no redo will be performed but a COMMIT signal will be sent. Whether or not the TRM recommends a redo is recorded in the Boolean variable *redo*. If so, and if *redo_counter* $=$ *max_redos*, an ABORT signal will be sent to the application. If *redo_counter* $<$ *max_redos*, the transaction will be re-executed by the TRM. As for the original transaction, the redo is performed in an atomic step, named PERFORM_REDO, and *redo_counter* is increased. After the redo, the TRM falls back to its "wait" location, and immediately leaves it with again invoking the checkers, since the condition "app_location = ta_performed" still is satisfied. The checkers will then check the redo for errors.

DFC and CFC model. Both checkers are invoked via synchronizing on the CHECK signal of the TRM (cf. Figures 5.5 and 5.6). The DFC checks all instructions where errors affect the data flow, i.e., all data-flow instructions. The CFC checks all instructions, since all instructions affect the control flow.

The DFC checks the data flow in one atomic step. If an error occurred in one of the data-flow instructions, it detects this error with probability *p_detn_DFC*. If no error occurred in the original data-flow instructions, an error might still be detected, i.e., the DFC has a false positive with probability *p_fp_DFC*.

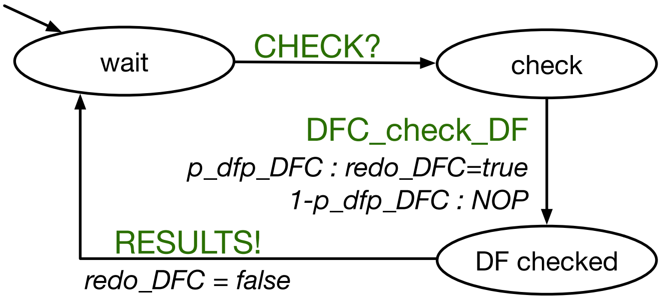

Figure 5.5: The program graph of the DFC. If an error occurred, $p\_dfp\_DFC = p\_detn\_DFC$. Otherwise $p\_dfp\_DFC = p\_fp\_DFC$. NOP means that no variables are changed.

To support a larger class of fault-tolerance techniques, we assume the control-flow checker to be able to check each type of instruction (control-flow, data-flow and transaction management) separately. We assume that all control-flow instructions are checked for errors first, followed by all data-flow instructions, and completed by checking all transaction management instructions. In each package, errors are detected with *p_detn_CFC_df*, *p_detn_CFC_cf*, and *p_detn_CFC_tmi*, respectively. Analogously, false positives occur with probabilities *p_fp_CFC_df*, *p_fp_CFC_cf*, and *p_fp_CFC_tmi*.

Nevertheless, configuration criteria used in Section 5.3 do not distinguish in the actions CFC_check_DF, CFC_check_CF, CFC_check_TMI or the states in-between. Thus in the exemplary configuration we can use the results of [Seg95] (Proposition 8.7.1 on probabilistic forward simulations) and replace the sequence of actions CFC_check_DF, CFC_check_CF, CFC_check_TMI by the single action CFC_check, representing the sequential execution of all three steps (cf. Figure 5.7).

When both checkers are finished, they synchronously send their results to the TRM and fall back to their "wait" location.

Hardware model. The hardware model keeps track of the state of the internal memory occupied by the application, i.e., it tracks errors and failures. It comes with two attributes: the probability of an error occurring in a single instruction, when the application is correct (p_e) and the increased probability of an error (p_e_incr) occurring in an instruction when the application has a failure, i.e., in a former transaction an undetectable error occurred.

The hardware is characterized by its internal status (operating normally, being erroneous, having a failure, or having a failure and another error occurred, cf Figure 5.8). Starting in location "correct" and synchronizing on both actions PERFORM_TA and PERFORM_REDO, an error occurs in the transaction with probability $p\_e\_ta = 1 - (1 - p\_e)^{ta\_len}$, leading to location "error". With probability $1 - p\_e\_ta$, the hardware stays in location "correct",

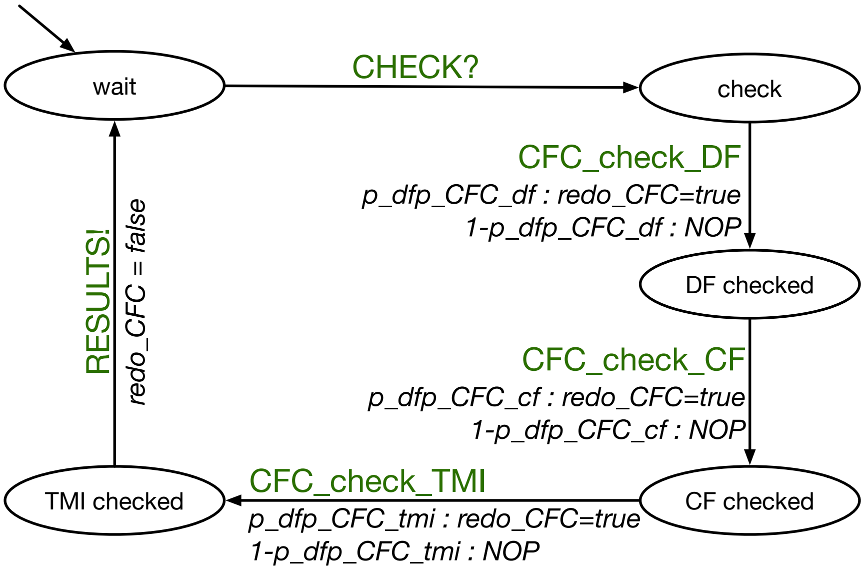

Figure 5.6: The program graph of the CFC. $p\_dfp\_CFC\_* = p\_detn\_CFC\_*$ if an error occurred, otherwise $p\_dfp\_CFC\_* = p\_fp\_CFC\_*$. NOP means that no variables are changed.

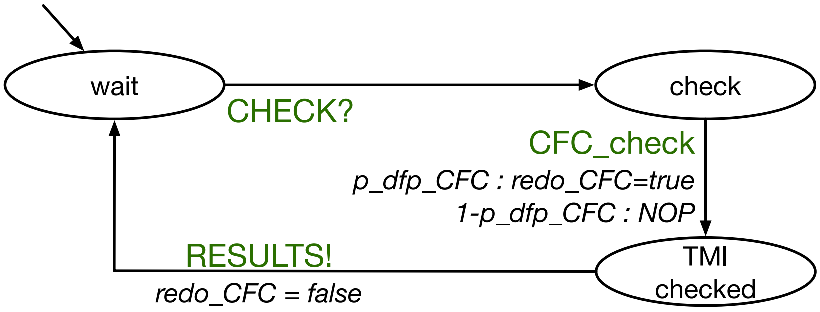

Figure 5.7: The program graph of the CFC after simplifying using simulation results. If an error occurred, $p\_\mathrm{dfp}\_\mathrm{CFC} = 1 - ((1 - p\_detn\_CFC\_df) \cdot (1 - p\_detn\_CFC\_cf) \cdot (1 - p\_detn\_CFC\_tmi))$, otherwise $p\_\mathrm{dfp}\_\mathrm{CFC} = 1 - ((1 - p\_fp\_CFC\_df) \cdot (1 - p\_fp\_CFC\_cf) \cdot (1 - p\_fp\_CFC\_tmi))$.

when synchronizing on PERFORM_TA or PERFORM_REDO. When the hardware is in location "error", and the TRM initiates a REDO, it falls back to the "correct" location. When a COMMIT signal is sent while the hardware in location "error", the error was not detected and thus the location is changed to "failure". Being in location "failure", the same behavior is modeled, but with increased error probabilities, and, when being in location "failure and error", both REDO and COMMIT change the location to "failure".

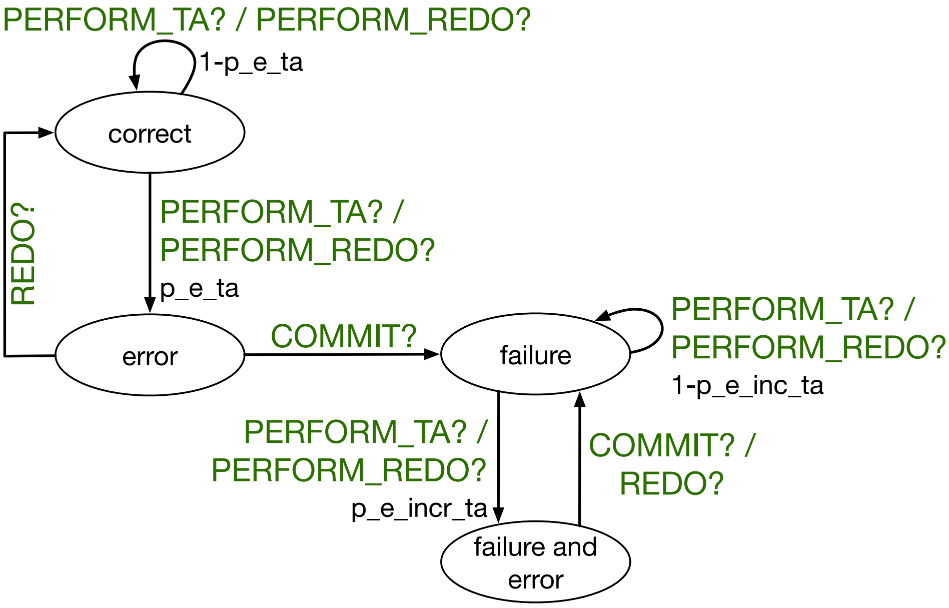
PERFORM_TA? / PERFORM_REDO?
1-p_e_ta
correct
REDO?
PERFORM_TA? / PERFORM_REDO?
p_e_ta
error
COMMIT?
failure
PERFORM_TA? / PERFORM_REDO?
1-p_e_inc_ta
PERFORM_TA? / PERFORM_REDO?
p_e_incr_ta
COMMIT? / REDO?
failure and error

Figure 5.8: The program graph of the hardware model.

component	**attribute**	**explanation**
Application	*cf_df_ratio*	percentage of control-flow instructions of *inst_num*
	inst_num	total number of instructions executed by the application, excluding duplicates
TRM	*max_redos*	number of transaction redos before application is aborted
	tmi_num	number of instructions added per transaction to implement the transaction mechanism
	ta_len	number of instructions per transaction
	p_e_TRM	probability of an error in the TRM

Figure 5.9: Summary of the model attributes (part 1).

component	**attribute**	**explanation**
DFC	p_detn_DFC	probability of error detection in data-flow instructions of a transaction
	p_fp_DFC	probability of false positive in data-flow instructions of a transaction
CFC	$p\_detn\_CFC\_df$	probability of detecting a control-flow error caused by some data-flow instruction
	$p\_detn\_CFC\_cf$	probability of detecting a control-flow error caused by some control-flow instruction
	$p\_detn\_CFC\_tmi$	probability of detecting a control-flow error caused by some transaction management instruction
	p_detn_CFC	combined probability of detecting control-flow error in a transaction
	$p\_fp\_CFC\_df$	probability of the CFC having a false positive when checking data-flow instructions of a transaction
	$p\_fp\_CFC\_cf$	probability of the CFC having a false positive when checking control-flow instructions of a transaction
	$p\_fp\_CFC\_tmi$	probability of the CFC having a false positive when checking transaction management instructions of a transaction
	p_fp_CFC	combined probability of the CFC having a false positive in a transaction
Hardware	p_e	probability of error in a single instruction
	p_e_incr	probability of error in a single instruction when the application has a failure

Figure 5.10: Summary of the model attributes (part 2).

5.3 Configuration of Redo-Based Fault Tolerance

In this section, we exemplarily configure an instance of our fault-tolerance model with respect to the system variables p_detn_CFC, p_detn_DFC, ta_len and max_redos. We utilize `fact` to compute rational functions, using ta_num as counter, exemplarily for three scenarios with $p\_e \in \{10^{-10}, 10^{-12}, 10^{-15}\}$.

Fault-Tolerance Setting. The model instance is mainly inspired by HAFT [Kuv+16], i.e., instructions in our model correspond to instructions on CPU-level, and error detection is enabled by duplicating instructions. The amount of duplicated instructions is configurable and represents the detection probability, e.g., replicating 80% of all data-flow instructions means $p\_detn\_DFC = 0.8$. Replicating all instructions gives error detection probabilities of 1, i.e., we neglect the probability of the same error occurring in both an original instruction and its replicate, which would cause the error to be undetectable. Furthermore we assume that transaction management instructions are not replicated.

Model Attributes. We fix the following model attributes: The application runs for exactly 10^{12} ($inst\_num$) instructions, 10% (cf_df_ratio) being control-flow instructions, and two (tmi_num) transaction management instructions ("begin of transaction" and "end of transaction") are inserted per transaction. The increased error probability is defined as $p\_e\_incr = (p\_e)^{\frac{8}{10}}$. False positives are errors that occur in replicated instructions or in transaction management instructions, i.e.,

$$\begin{aligned}
p\_fp\_CFC = 1 - \Big(& (1 - p\_e)^{ta\_len \cdot cf\_df\_ratio \cdot p\_detn\_CFC} \\
& \cdot (1 - p\_e)^{ta\_len \cdot (1 - cf\_df\_ratio) \cdot p\_detn\_DFC} \\
& \cdot (1 - p\_e)^{tmi\_num} \Big) \qquad \text{and} \\
p\_fp\_DFC = 1 - \Big(& (1 - p\_e)^{ta\_len \cdot (1 - cf\_df\_ratio) \cdot p\_detn\_DFC} \Big).
\end{aligned}$$

Reward Structures. In the configuration, we focus on the expected time overhead arising through error detection and correction. For this, we introduce a reward structure that assigns one time unit each time an instruction is executed for error detection or correction. Formally, we define a reward structure assigning states where the TRM is in location "answers received" the following values:

$$\begin{aligned}
\text{If } \mathsf{redo\_counter} = 0\text{: } & ta\_len \cdot p\_detn\_CFC \\
& + ta\_len \cdot cf\_df\_ratio \cdot p\_detn\_DFC \\
& + tmi\_num \qquad \text{and} \\
\text{if } \mathsf{redo\_counter} > 0\text{: } & ta\_len \cdot p\_detn\_CFC \\
& + ta\_len \cdot cf\_df\_ratio \cdot p\_detn\_DFC \\
& + tmi\_num \\
& + ta\_len,
\end{aligned}$$

and 0 to all other states. The reward assignment depends on whether an original transaction or one of its redos was executed. If an original transaction was executed, the assigned reward corresponds to the number of duplicated instructions. If a redo was executed, the assigned reward is the number of original instructions plus the number of duplicated instructions.

Configuration Criteria. The goal of this section is to exemplarily find good parameter values that optimize the chosen protocol instance with respect to the following criteria:

1. The probability of terminating correctly shall be at least 0.9995,
2. the conditional probability of aborting, in case of not terminating correctly, shall be greater than 0.15, and
3. from all configurations meeting the above conditions, one with minimal overhead is chosen.

Finding the Optimal Configuration. We exemplarily configure the fault-tolerance model for all $p\_e \in \{10^{-10}, 10^{-12}, 10^{-15}\}$. For each $max\_redos \in \{0, 1, 2, 3\}$ we consider a discrete number of combinations for the detection probabilities

$$p\_detn\_DFC, p\_detn\_DFC \in \{0, 0.001, 0.1, 0.5, 0.75, 0.9, 0.95, 0.99, 0.999\}$$

and for the transaction lengths

$$ta\_len \in \{100, 200, 500, 1000, 2000, 5000, 10^4, 10^6, 10^{10}, 10^{12}\}$$

to fill decision tables and plot the rational functions. For each error probability, we use these decision tables to find the optimal configuration in the selected ones.

The system variables p_detn_CFC and p_detn_DFC are attributes that arise only in transition probabilities and reward assignments, and hence can be used as parameters. We also handle the error probability p_e as parameter.

For each sub-model that represents one transaction, ta_len is also an attribute arising only in transition probabilities and rewards. Nevertheless, since $ta\_num = inst\_num/ta\_len$, ta_len influences the maximal value of the counter ta_num and hence the model structure.

We apply counter-based factorization (see Chapter 4) using ta_num as counter. Thus, the factors of the redo-based fault-tolerance model represent exactly one transaction. Therefore, ta_len can be handled parametric when computing local characteristics. In the global rational function, ta_len is present in the form $inst\_num/ta\_len$, since this value is the maximal value of ta_num, arising in the sum indices and matrix exponents.

Using the counter-based factorization approach we compute matrices and vectors $\mathscr{P}$, $\mathscr{E}$, $\mathscr{P}^{(0)}$, $\mathscr{E}^{(0)}$, $\mathfrak{p}^{(n)}$, and $\mathfrak{e}^{(n)}$ for each $max\_redos \in \{0, 1, 2, 3\}$, and thus invoke probabilistic model checking only once for each maximal number of redos. The matrices and vectors are parameterized over the error probability, detection probabilities, and the transaction length.

To systematically explore the design space, we first fix the error probabilities and replace the parameters with constants within the vectors and matrices, as these values can be assumed to be given.

The Maximal Number of Redos. We start with analyzing the effect of the maximal number of redos. Without any fault-tolerance mechanism[1], the probability of terminating correctly is $3 \cdot 10^{-83}$ for $p\_e = 10^{-10}$, 0.15 for $p\_e = 10^{-12}$, and for $p\_e = 10^{-15}$ it is 0.998. Figures 5.11 and 5.12 show the probability of terminating correctly, when varying *max_redos*. In Figure 5.11, a transaction lengths of 1000 is fixed, and the probability of terminating correctly in dependence of the detection rates is depicted. Figure 5.12 depicts this probability in dependence of the transaction lengths, when fixing the detection probabilities to be $p\_detn\_DFC = p\_detn\_DFC = 0.9$.

For all chosen error probabilities and detection probabilities, performing a single redo pays off drastically. For example, consider the setting where the transaction length is 1000 and the detection probabilities both are set to 0.9. Then, allowing a single redo increases the probability of terminating correctly from 0.027 to 0.827, when $p\_e = 10^{-12}$, and from 0.996 to 0.9998 when $p\_e = 10^{-15}$.

When performing error detection without redo-based correction, increasing detection probabilities causes the probability of terminating correctly to shrink, since more instructions are replicated and thus more replicas can be affected by errors. Performing redos neglects this effect. Also for all transaction lengths, except for the extrem case where the whole application is a single transaction, performing redos significantly increases the probability of terminating correctly. Allowing more than one redo increases the probability of terminating correctly only marginally, except for transaction lengths above 10^{10}. Figures 5.13 and 5.14 visualize the conditional probabilities of aborting when not terminating correctly. As before, we fix $ta\_len = 1000$ in Figure 5.13 and $p\_detn\_DFC = p\_detn\_DFC = 0.9$ in Figure 5.14. The figures show that each redo decreases the chance of aborting in case of not terminating correctly (criterion 2). For two redos, this chance is almost zero for all configurations except for extremely large transaction lengths and small error probabilities.

The correlation of the overhead and the maximal number of redos is depicted in Figures 5.15 and 5.16. For low error probabilities, the overhead is only marginally affected by the maximal number of redos, since, when errors are unlikely and thus error correction is invoked only seldomly, the overhead is mainly caused by executing duplicated instructions for error detection. For higher error probabilities, expectably more errors occur and thus more error correction needs to be performed. Consequently, the overhead increases when allowing more redos. Thus, choosing to perform at most one redo increases the probability of correct termination. Allowing another redo does not significantly increase this probability, but decreases the probability of aborting in case of not terminating correctly without decrease in the overhead. Hence, from now on we fix $max\_redos = 1$.

Optimal Transaction Lengths. Figure 5.17 shows the probability of terminating correctly for varying transaction lengths and detection probabilities, when fixing $max\_redos = 1$. For $p\_e = 10^{-10}$, the probability of terminating correctly is hardly affected by varying transaction lengths below 10^6. Choosing longer transaction lengths decreases the probability substantially. Large transaction lengths do increase the probability again, when *p_detn_DFC* is small, but short transaction lengths are in general to be preferred. For lower error probabilities, the turning point moves to the right. For error probability 10^{-15} it is beyond the maximal possible transaction length.

---

[1] Due to the nature of our model and PMC, these values need to be computed in separate runs. Applying counter-based factorization, this took less than three seconds.

The conditional probability of aborting when not terminating correctly (cf. Figure 5.18) first increases with increasing transaction lengths, then stays on a level near 1 for middle-large transaction lengths and finally drops again, in the same point as the probability of terminating correctly rises for some detection probabilities. Again, the turning points move to the right when decreasing error probabilities.

Regarding the overhead, small transaction lengths cause less transactions to be erroneous and thus less error correction to be performed. Nevertheless, very small transaction lengths cause error detection to take place very often and thus lead to a higher overhead. This is visible in Figure 5.19. For error probabilities 10^{-10} and 10^{-12}, increasing the transaction length up to 10^{10} decreases the overhead, but after this point, the overhead increases significantly. For error probability 10^{-15} the effect is barely visible. The decrease of the overhead also arises because the probability of aborting increases with higher transaction length (cf. Figure 5.20). The higher this probability, the sooner the application is likely to be aborted. After an abort no more overhead is produced. Thus, the overhead drops with increasing abort rates and rises when very large transaction lengths cause a shrinking abort rate. For $p\_e = 10^{-10}$, $ta\_len = 10^6$ is optimal among the investigated lengths, independent from the detection probabilities. For $p\_e = 10^{-12}$, we choose $ta\_len = 10^{10}$, and for $p\_e = 10^{-15}$, errors are unlikely enough to handle the application as one transaction.

Error Detection Probabilities. We now fix the remaining configuration parameters *p_detn_CFC* and *p_detn_DFC*. Figure 5.21 shows the effect of these parameters on the probability of terminating correctly, with the previously chosen configurations for *max_redos* and *ta_len*. Figure 5.22 shows the probability of aborting in case of not terminating correctly, and in Figure 5.23 the effect on the expected overhead can be seen. As expected, all three values increase when increasing the detection probabilities. Increasing only one error detection probability certainly has a visible effect on the probability of terminating correctly, yet it is ineffective to choose one detection probability to be very low and the other one to be very high. To configure the remaining parameters of the fault-tolerance technique, we use a decision table, exemplarily for $p\_e = 10^{-15}$. Table 5.24 shows results for $max\_redos = 1$ and $ta\_len = 10^{12}$. All depicted lines satisfy the first two configuration criteria. For $p\_e = 10^{-15}$, the configuration satisfying the configuration criteria is $max\_redos = 1$, $ta\_len = 10^{12}$, $p\_detn\_DFC = 0.99$, and $p\_detn\_CFC = 0.95$.

The overhead of the depicted configurations does not differ much, but the conditional probability of aborting when not terminating correctly can be increased significantly when accepting a little more overhead. Thus, it would be worth choosing a configuration that replicates some more instructions, accepting a little more overhead, e.g., $p\_detn\_CFC = p\_detn\_DFC = 0.999$.

Measurement Times and Size of the Rational Functions. We applied the factorization approach to the redo-based fault-tolerance model and thus retrieved the rational functions in “matrix-based” representation. The matrix-based representation of a rational function is the respective formula presented in Corollary 4.17, without actually applying a computer algebra system to compute the result of the formula. An explicit representation of the formula can be obtained by using sympy to compute the result of the formulas in Corollary 4.17. This matrix-based representation is much smaller than some explicit representation of the rational function. Actually, an explicit representation for the rational functions considered in this chapter could not be computed due to the enormous size. To give the reader an impression of the size, we computed some explicitly represented rational functions for short applications. For a model with $\mathit{max\_redos} = 0$ and $\mathit{ta\_num} = 10$, the rational function in the explicit representation computed by sympy consumed, in text representation, 4.7 megabyte of memory. For $\mathit{max\_redos} = 0$ and $\mathit{ta\_num} = 1000$, the disk space needed was 1200 megabyte. Exporting this rational function to a text file took more than a day.

Thus, to obtain point-wise evaluations of the rational functions, we did not compute the explicit representations. Instead, we appointed concrete values directly to the local rational functions in the matrices, and used sympy afterwards to obtain the respective evaluation.

Experiments were performed single-threaded on a machine with a 2.5 GHz Intel Core i7 CPU and 16 GB RAM single-threaded. The computation of all matrices and vectors $\mathscr{P}$, $\mathscr{E}$, $\mathscr{P}^{(0)}$, $\mathscr{E}^{(0)}$, $\mathfrak{p}^{(n)}$, and $\mathfrak{e}^{(n)}$ took 50 seconds for $\mathit{max\_redos} = 0$, 173 seconds for $\mathit{max\_redos} = 1$, 140 minutes for $\mathit{max\_redos} = 2$, and about one day and three hours for $\mathit{max\_redos} = 3$. Evaluating the rational functions point-wise using the matrix-based representation to set up decision tables took less than a second per evaluation point for $\mathit{max\_redos} = 0$ and about 3 seconds per point for $\mathit{max\_redos} = 3$.

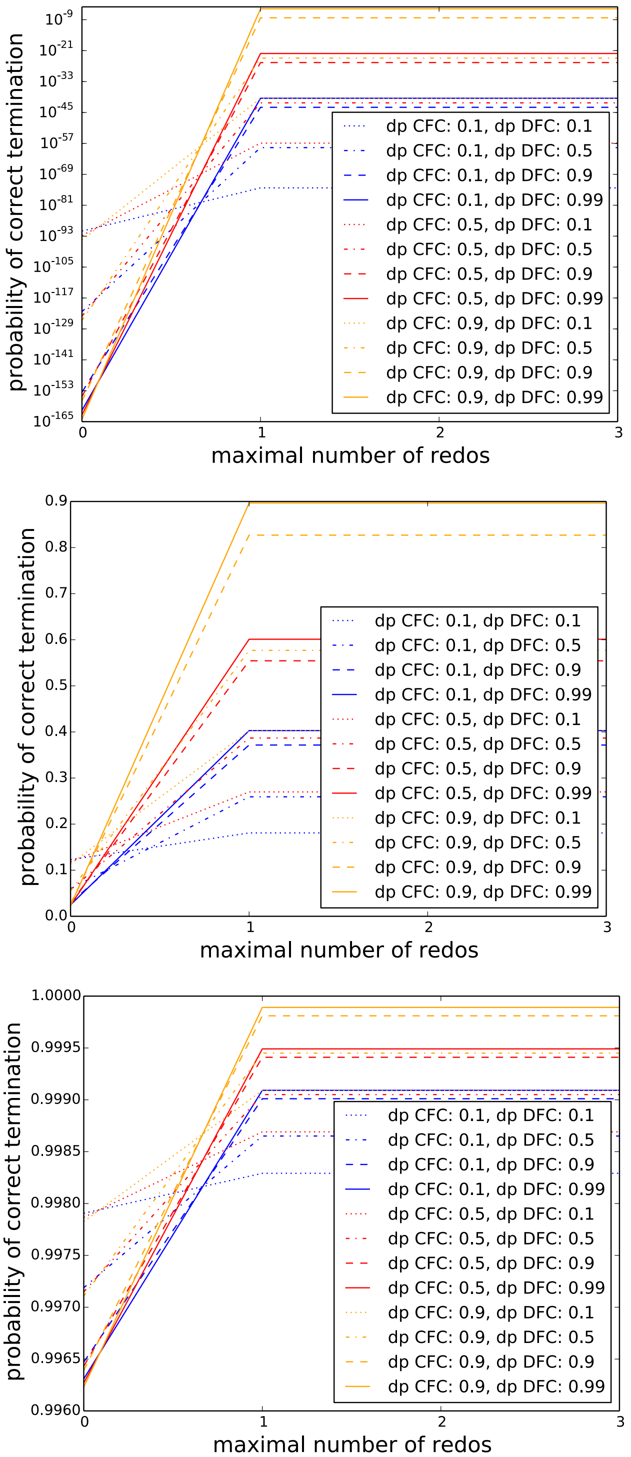

Figure 5.11: Probability of terminating correctly in dependence on the maximal number of redos, when varying the detection probabilities and fixing $ta\_len = 1000$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$. Note that the y-scale in the topmost picture is in log scale.

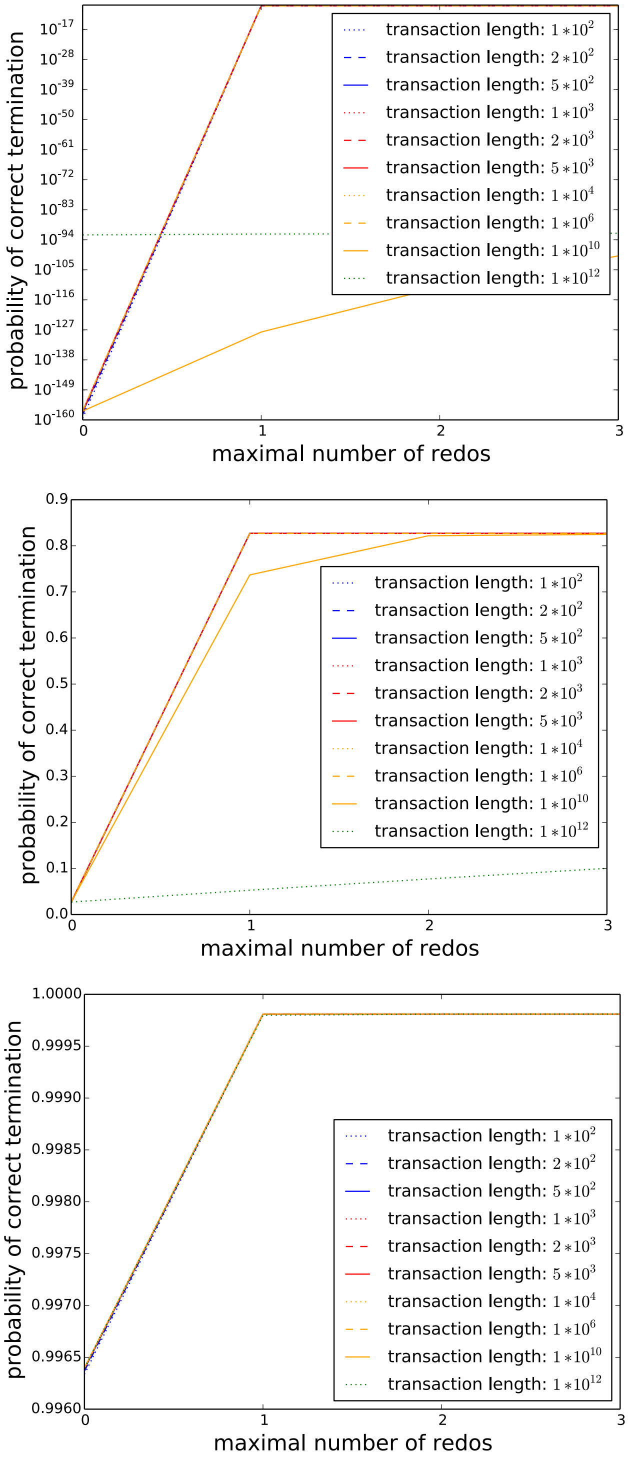

Figure 5.12: Probability of terminating correctly in dependence on the maximal number of redos, when varying the transaction length and fixing $p\_detn\_CFC = p\_detn\_DFC = 0.9$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$. Note that the y-scale in the topmost picture is in log scale.

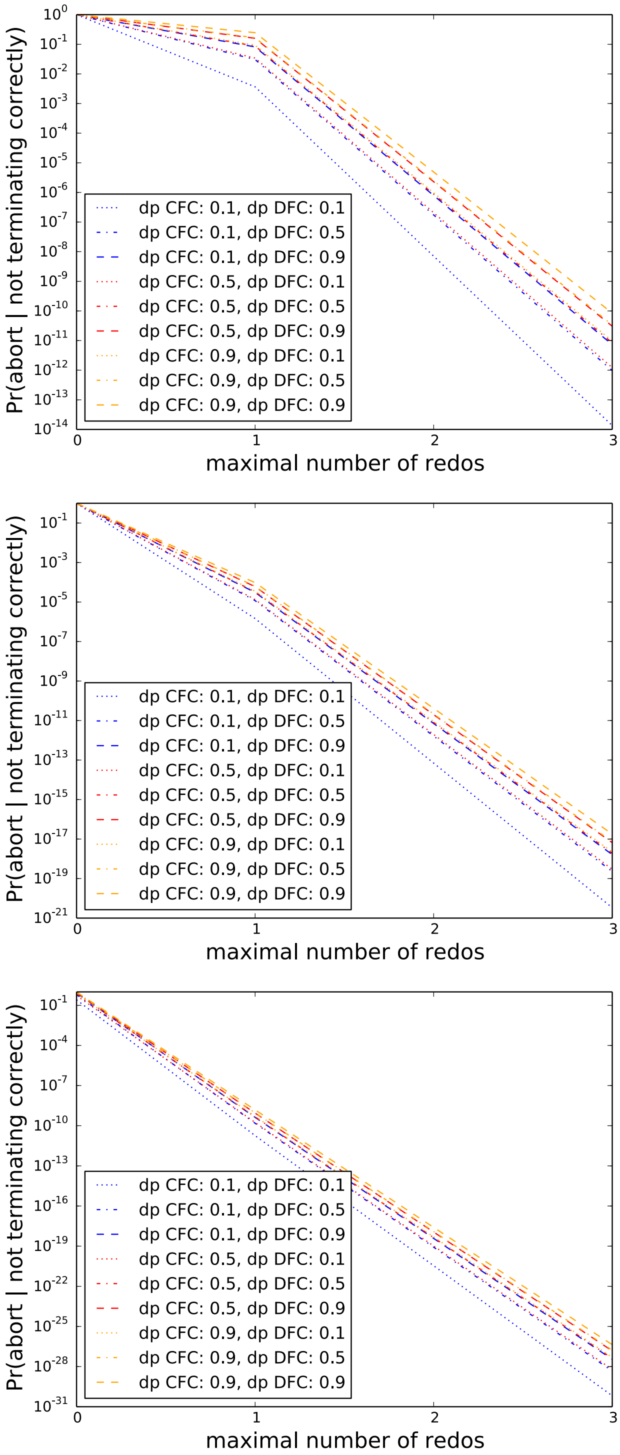

Figure 5.13: Conditional probability of aborting in case of not terminating correctly in dependence of the maximal number of redos, when varying the detecting probabilities and fixing *ta_len* $= 1000$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$. Note that y-scales are in log scale.

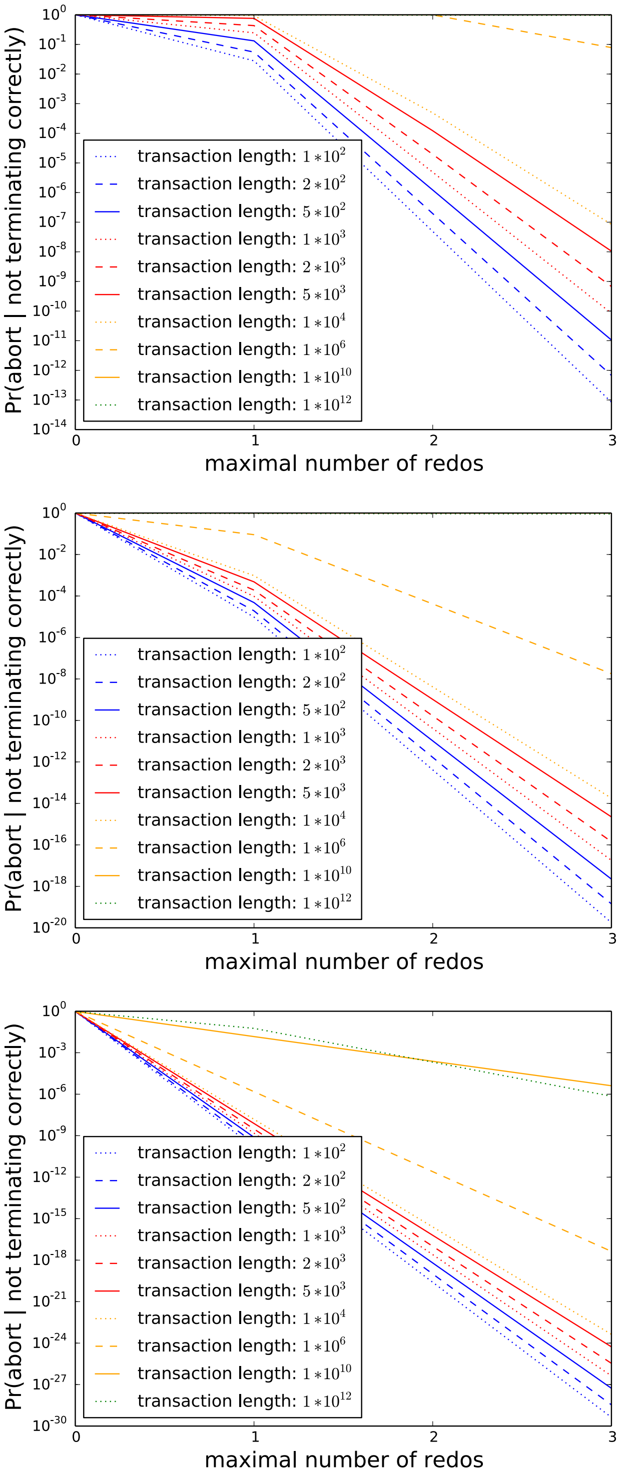

Figure 5.14: Conditional probability of aborting in case of not terminating correctly in dependence of the maximal number of redos, when varying the transaction length and fixing $p\_detn\_CFC = p\_detn\_DFC = 0.9$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$. Note that y-scales are in log scale.

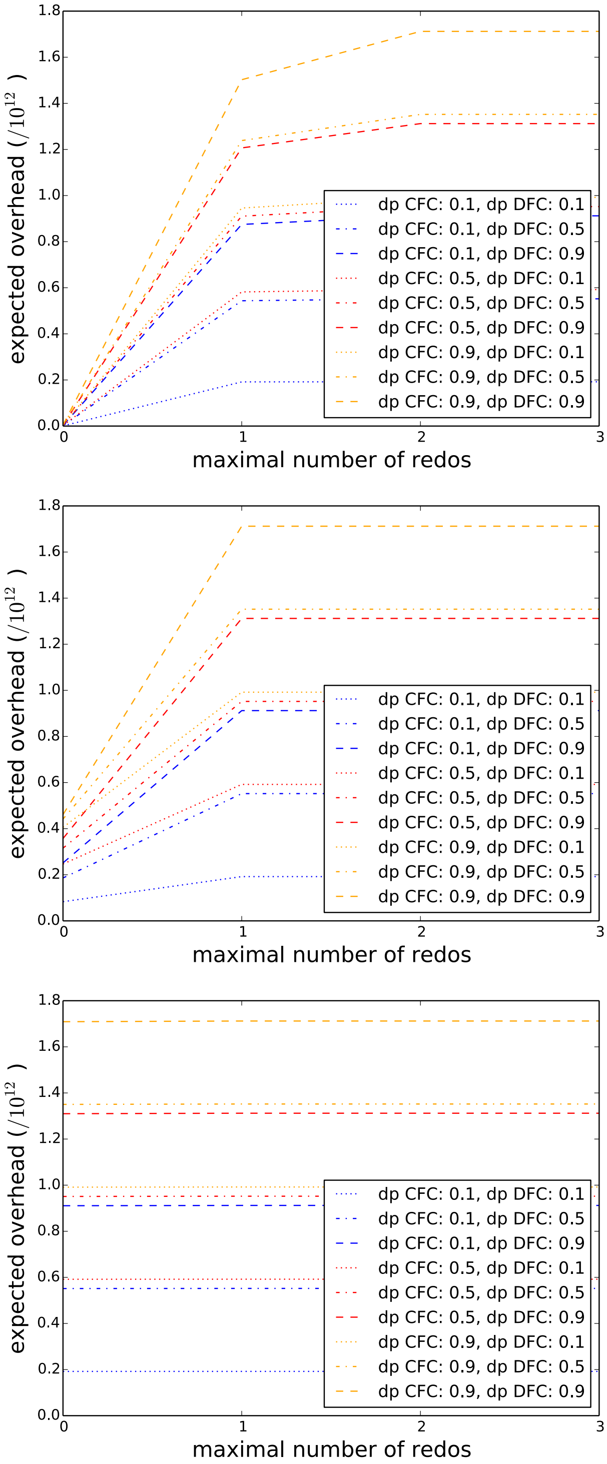

Figure 5.15: Expected overhead in dependence of the maximal number of redos, when varying the detection probabilities and fixing $ta\_len = 1000$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

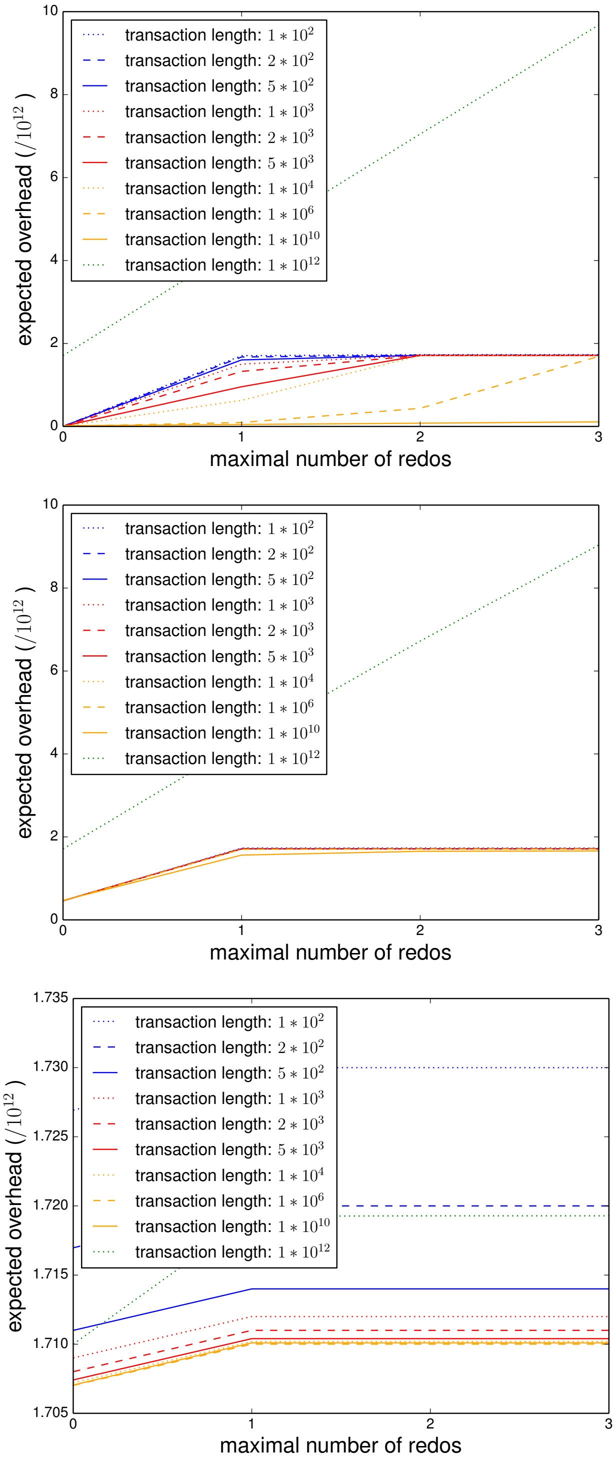

Figure 5.16: Expected overhead in dependence of the maximal number of redos, when varying the transaction length and fixing $p\_detn\_CFC = p\_detn\_DFC = 0.9$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

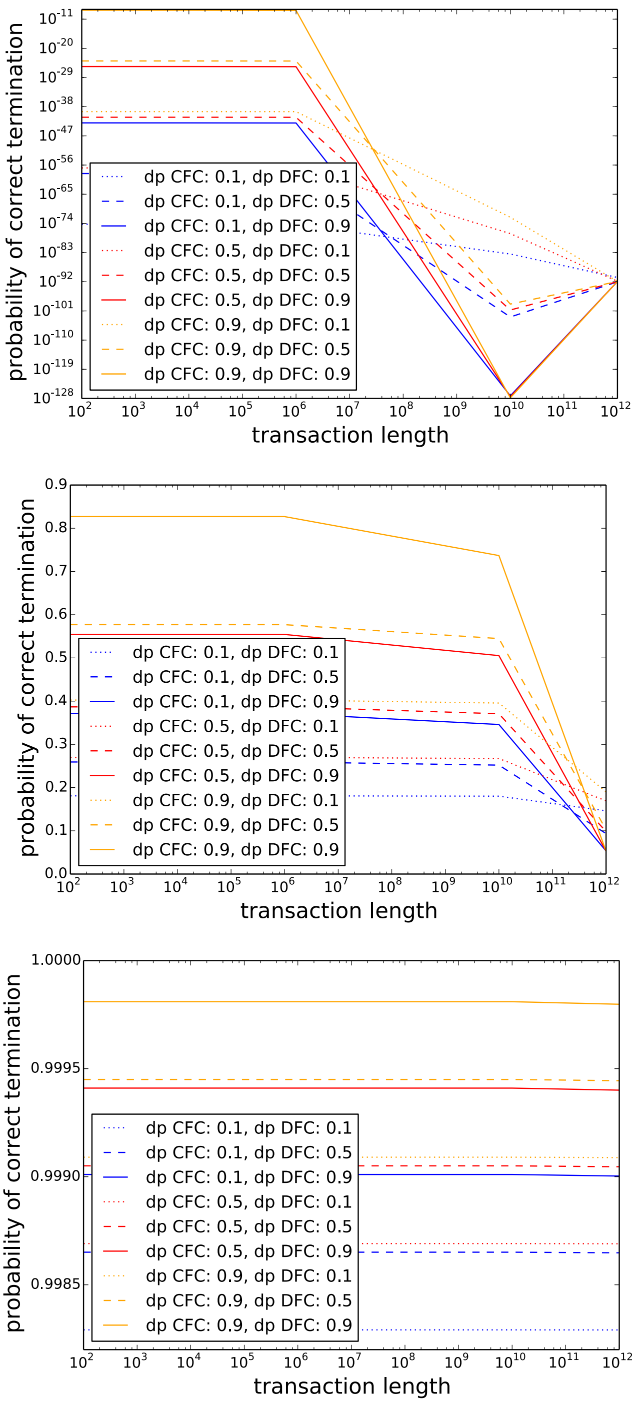

Figure 5.17: Probability of terminating correctly in dependence of the transaction length, when fixing $max\_redos = 1$ and varying the detection probabilities. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

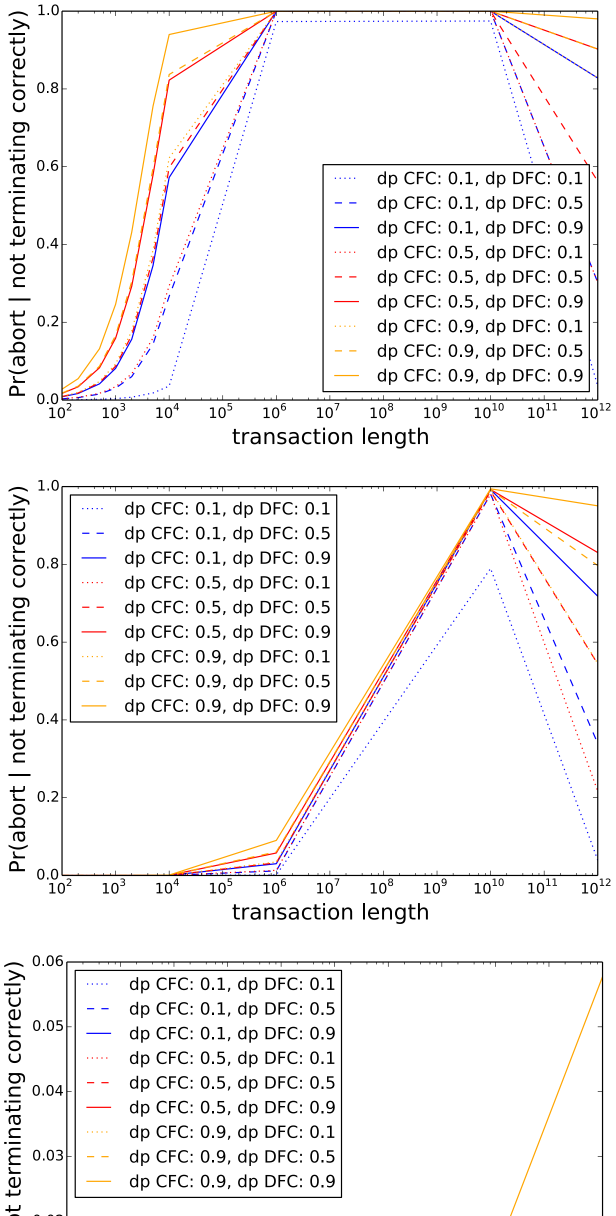

Figure 5.18: Conditional probability of aborting in case of not terminating correctly in dependence of the transaction length, when fixing $max\_redos = 1$ and varying the detection probabilities. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

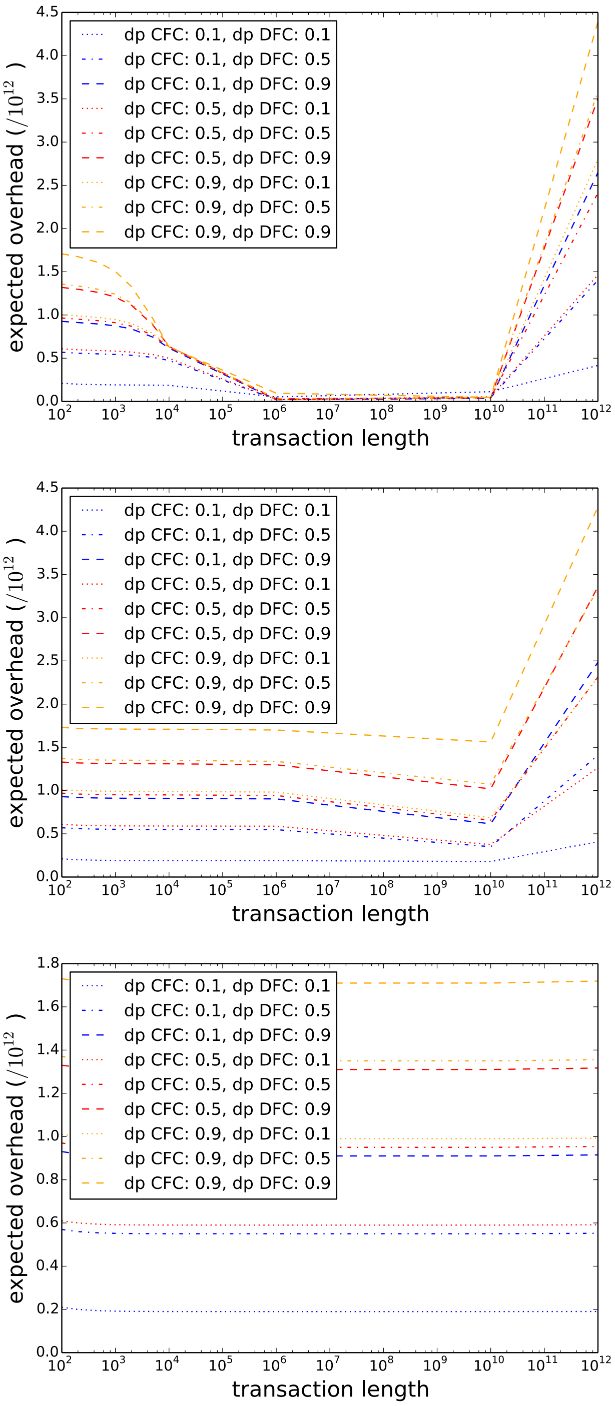

Figure 5.19: Expected overhead in dependence of the transaction length, when fixing *max_redos* = 1 and varying the detection probabilities. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

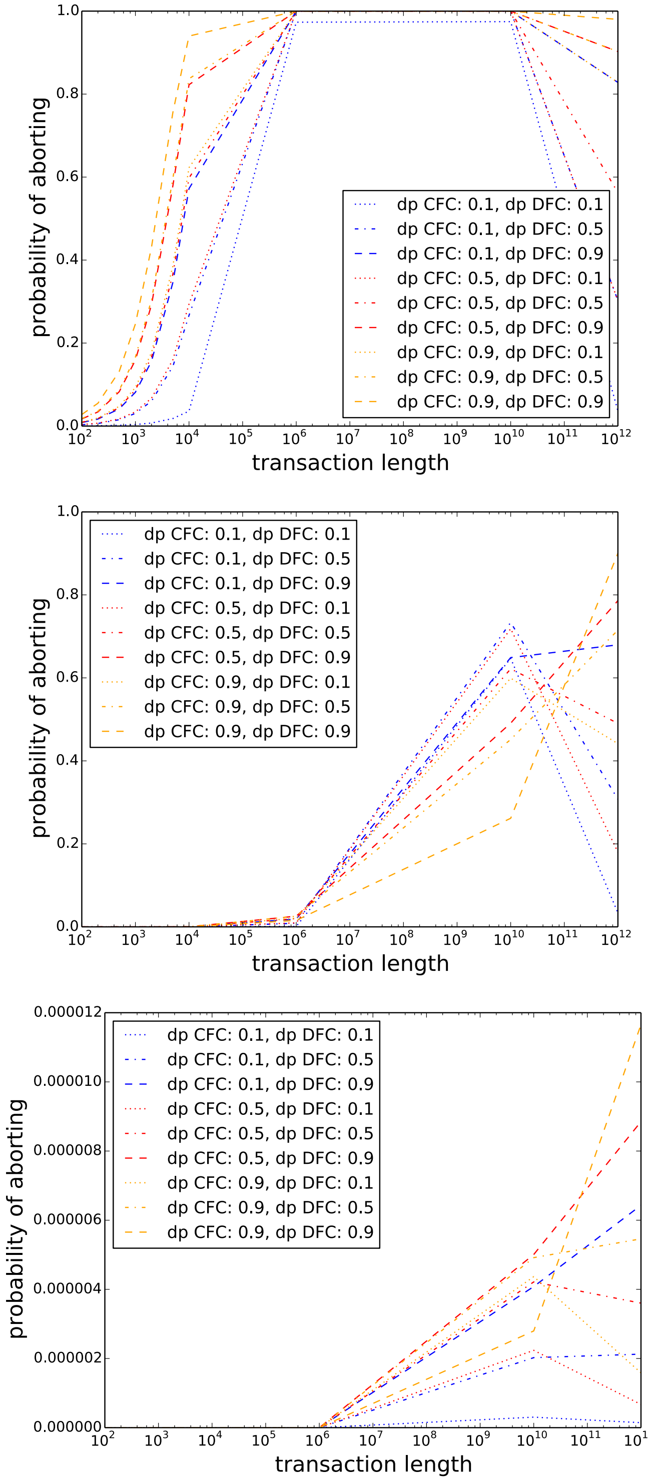

Figure 5.20: The (unconditional) probability of aborting in dependence of the transaction length for ranging detection probabilities, when $max\_redos = 1$. From top to bottom: error probability $10^{-10}, 10^{-12}, 10^{-15}$.

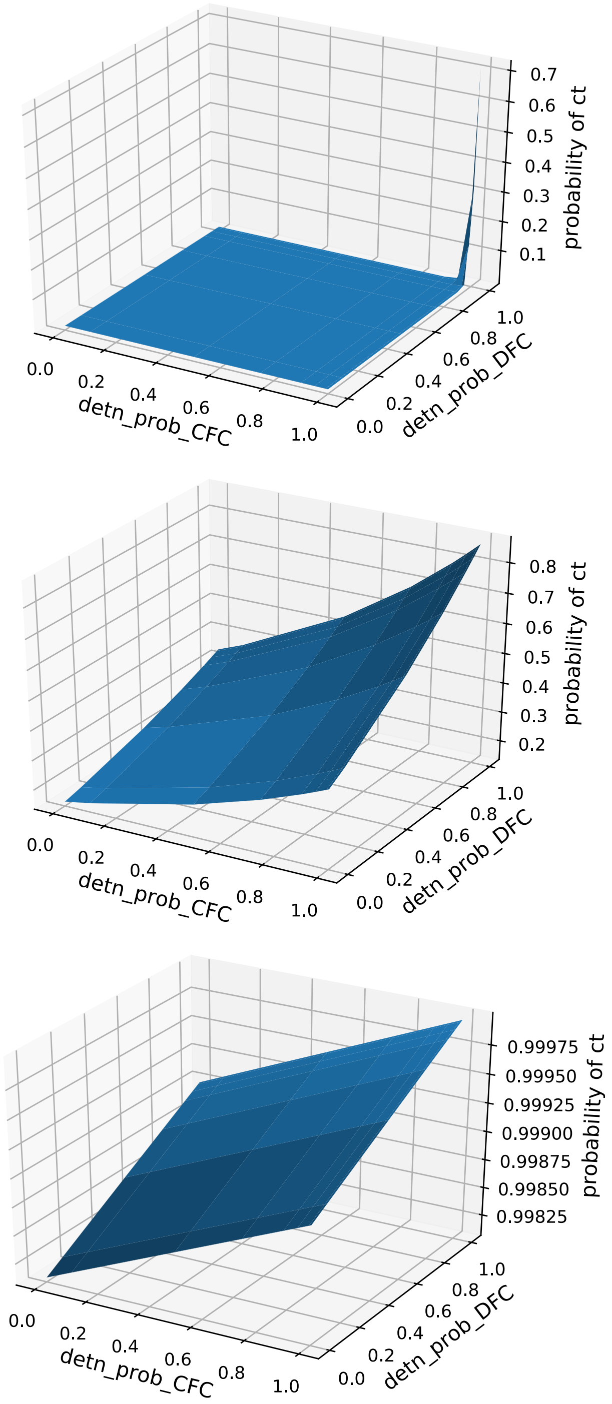

Figure 5.21: Probability of terminating correctly in dependence of the detection probabilities, when fixing $max\_redos = 1$. From top to bottom: $p\_e = 10^{-10}$ and $ta\_len = 10^6$, $p\_e = 10^{-12}$ and $ta\_len = 10^{10}$, $p\_e = 10^{-15}$ and $ta\_len = 10^{12}$. “ct” is short for correct termination.

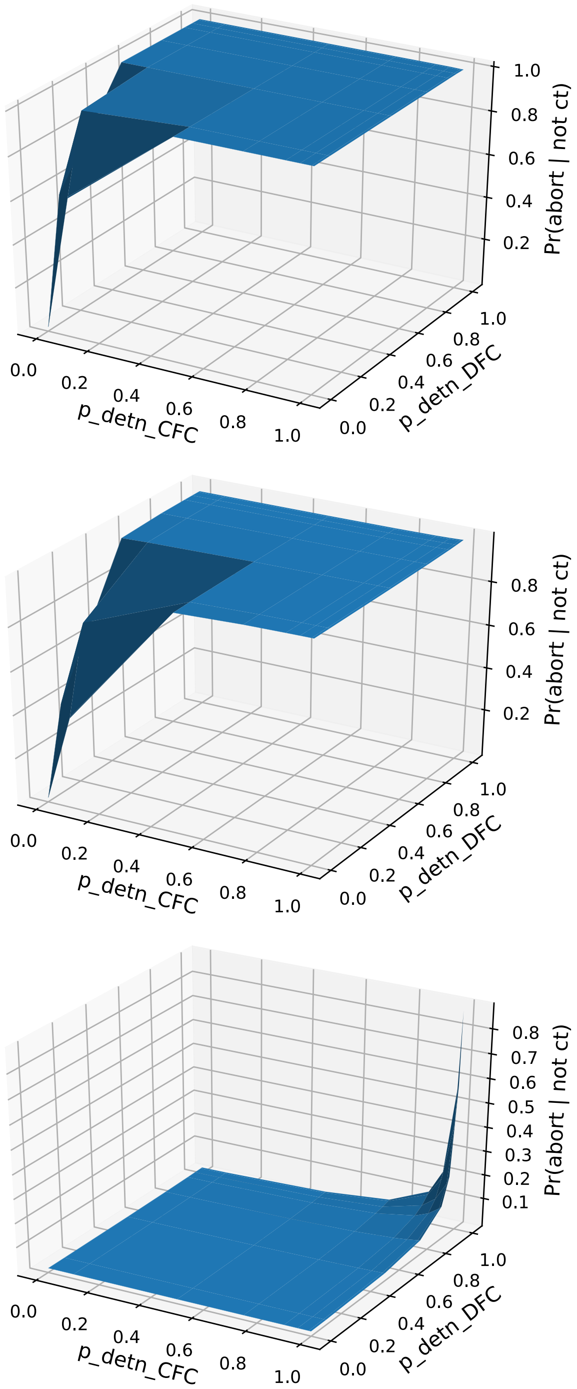

Figure 5.22: Conditional probability of aborting in case of not terminating correctly in dependence of the detection probabilities, when fixing $max\_redos = 1$. From top to bottom: $p\_e = 10^{-10}$ and $ta\_len = 10^{6}$, $p\_e = 10^{-12}$ and $ta\_len = 10^{10}$, $p\_e = 10^{-15}$ and $ta\_len = 10^{12}$. “ct” is short for correct termination.

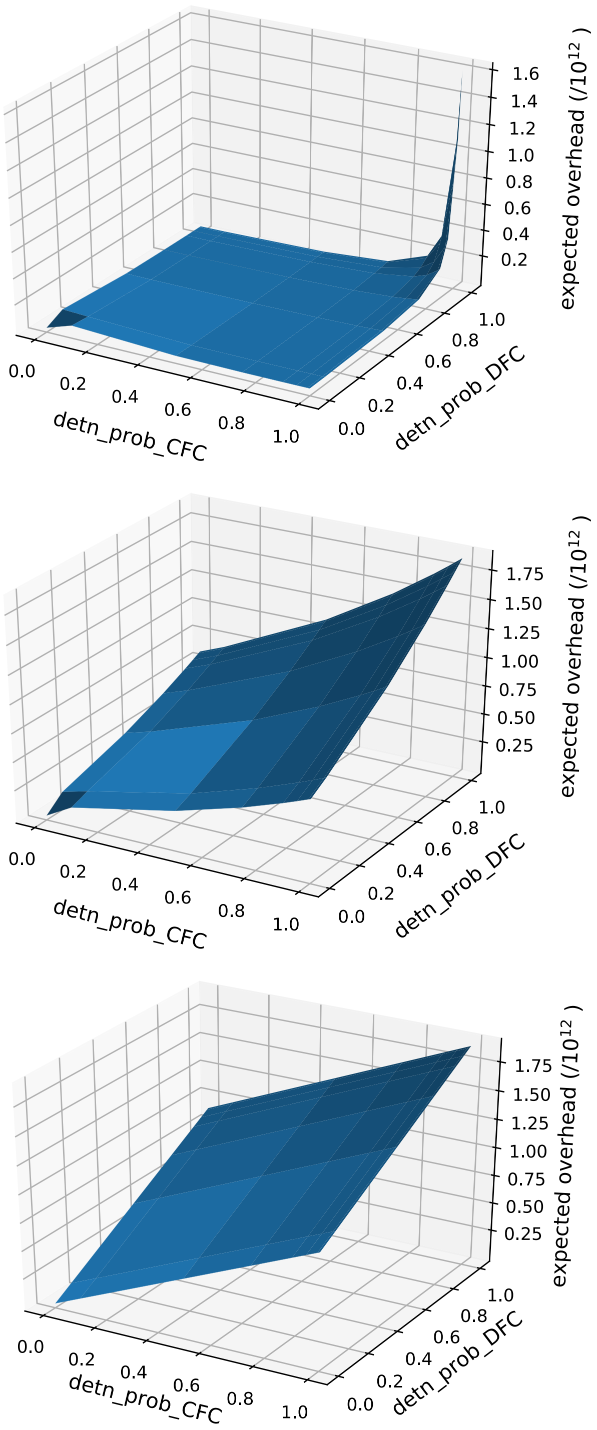

Figure 5.23: Expected overhead in dependence of the detection probabilities, when fixing $max\_redos = 1$. From top to bottom: $p\_e = 10^{-10}$ and $ta\_len = 10^6$, $p\_e = 10^{-12}$ and $ta\_len = 10^{10}$, $p\_e = 10^{-15}$ and $ta\_len = 10^{12}$.

p_detn_CFC	p_detn_DFC	Criterion 1	Criterion 2	Criterion 3
0.95	0.99	0.99992	0.19	$1.851 \cdot 10^{12}$
0.95	0.999	0.99994	0.21	$1.86 \cdot 10^{12}$
0.99	0.95	0.99993	0.19	$1.855 \cdot 10^{12}$
0.99	0.99	0.99997	0.43	$1.892 \cdot 10^{12}$
0.99	0.999	0.99997	0.57	$1.9 \cdot 10^{12}$
0.999	0.95	0.99994	0.23	$1.864 \cdot 10^{12}$
0.999	0.99	0.99998	0.59	$1.901 \cdot 10^{12}$
0.999	0.999	0.99998	0.88	$1.909 \cdot 10^{12}$

Figure 5.24: Decision table for $p\_e = 10^{-15}$, $ta\_len = 10^{12}$ and $max\_redos = 1$.

6 Conclusion and Further Work

This thesis studies the configuration of fault-tolerant systems with formal methods, in particular probabilistic model checking. We investigated strengths and weaknesses and exemplarily presented the systematic configuration processes. This work hopefully encourages the more frequent usage of formal methods, especially in the field of fault-tolerance configuration.

Inter-Process Communication. We showed how a domain-specific language and a model generation tool facilitate the modeling process and reduce the risk of human mistakes. We generated a model for a concrete set of communicating processes forming a space probe, and we showed that this model suffers from well-known inaccuracy and time issues of iterative solvers. We revealed a correlation between these issues and extremely small or large transition probabilities. This further motivated the usage of non-iterative methods, in addition to the argument that non-iterative methods can handle parametric models. Then, we used PMC and Newton's method to find the optimum of the space probe's system variable that maximized its availability.

Redo-Based Fault Tolerance. We presented a model for redo-based fault-tolerance techniques that can be instantiated to a concrete technique by specifying attributes. The fault-tolerance mechanism was inspired by HAFT [Kuv+16]. The state-explosion problem arising in this model due to the large size of the protected application was tackled with a new approach, counter-based factorization. Yet, the configuration process of this model revealed that rational functions that are obtained with PMC, when not being represented by matrices, can be huge — too huge to analyze them with mathematical methods, as we did for the space probe model. Therefore, we used the matrix-based representation to obtain plots and point-wise evaluations of the chosen configuration criteria. Since in this protocol multiple system variables needed to be configured, we exemplarily presented how to systematically explore the design space to retrieve the target configuration.

Counter-Based Factorization. Counter-based factorization is a new promising approach to tackle the state-explosion problem for DTMCs that comprise counters. Although only applied to the redo-based fault-tolerance model in this thesis, the results give hope that also for other models computation times can be reduced drastically — for both the parametric and non-parametric setting.

Further Work

Factorization. Interesting further work would be to extend `fact` such that it supports other counters than simple, observing ones. Furthermore, support for multiple counters is intended. Other future work is the extension of the counter-based factorization approach to arbitrary ω-regular properties, steady-state properties, and other model types. Applying factorization to other models will reveal whether this approach is of large impact beyond the redo-based fault-tolerance protocol. This will probably also reveal new challenges and new possibilities of extending the factorization approach.

Fault-Tolerance Configuration. It will be beneficial to investigate the pros and cons of formal methods for other fault-tolerance protocols than the chosen two, especially those that do not have a DTMC as underlying model. For example, for non-deterministic models new challenges may arise. It also would be beneficial to further investigate the correlation of small transition probabilities and the inaccuracy and time issues of iterative methods. This work demonstrated that these issues do not only arise in some theoretic examples but are relevant for real-world models. Yet, it is not clear under which precise circumstances these issues arise, and whether there are iterative solutions that can prevent both the inaccuracy and high time consumption.

Retrieving rational functions from PMC is beneficial as long as these functions are relatively small. For large functions we need shorter representations, like the matrix-based representation, and optimization methods that work directly on these short representations. Investigating such representations and methods will also be interesting future work.

Bibliography

[11a] *Even Single Events Can Be Very Upsetting.* 2011. URL: `http://lasp.colorado.edu/home/maven/2011/01/07/even-single-events-can-be-very-upsetting/`.

[11b] *The* PRISM *Language.* 2011. URL: `http://prismmodelchecker.org/manual/ThePRISMLanguage/Introduction`.

[16a] *Google Cloud Storage Service Level Agreement.* 2016. URL: `https://cloud.google.com/storage/sla`.

[16b] *L4/Fiasco.OC microkernel.* 2016. URL: `https://os.inf.tu-dresden.de/fiasco/overview.html`.

[18a] *Amazon Compute Service Level Agreement.* 2018. URL: `https://aws.amazon.com/de/compute/sla/`.

[18b] *Lark - A modern parsing library for Python.* 2018. URL: `https://lark-parser.readthedocs.io/en/latest/`.

[Abd+18] J. Abdullah, G. Dai, M. Mohaqeqi, and W. Yi. “Schedulability Analysis and Software Synthesis for Graph-Based Task Models with Resource Sharing”. In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium.* 2018, pp. 261–270. DOI: `10.1109/RTAS.2018.00034`.

[Afl+17] S. Aflaki, M. Volk, B. Bonakdarpour, J. Katoen, and A. Storjohann. “Automated Fine Tuning of Probabilistic Self-Stabilizing Algorithms”. In: *2017 IEEE 36th Symposium on Reliable Distributed Systems.* 2017, pp. 94–103. DOI: `10.1109/SRDS.2017.22`.

[AH16] W. Ahmad and O. Hasan. “Formal Availability Analysis Using Theorem Proving”. In: *Formal Methods and Software Engineering - 18th International Conference on Formal Engineering Methods.* 2016, pp. 226–242. DOI: `10.1007/978-3-319-47846-3_15`.

[AHJ01] L. de Alfaro, T. A. Henzinger, and R. Jhala. “Compositional Methods for Probabilistic Systems”. In: *International Conference on Concurrency Theory.* Ed. by K. G. Larsen and M. Nielsen. Springer Berlin Heidelberg, 2001, pp. 351–365. ISBN: 978-3-540-44685-9. DOI: `10.1007%2F3-540-44685-0_24`.

[Ahm+14] W. Ahmad, O. Hasan, S. Tahar, and M. S. Hamdi. “Towards the Formal Reliability Analysis of Oil and Gas Pipelines”. In: *International Conference on Intelligent Computer Mathematics* 8543 (2014). DOI: `10.1007/978-3-319-08434-3_4`.

[AHS12] T. Adam, U. Hashim, and U. S. Sani. “Retrieving the Correct Information: Channel Coding Reliability in Error Detection and Correction”. In: *International Conference on Computational Intelligence, Modelling and Simulation*. 2012, pp. 400–404. DOI: `10.1109/CIMSim.2012.90`.

[AHT15] W. Ahmad, O. Hasan, and S. Tahar. “Formal reliability analysis of wireless sensor network data transport protocols using HOL”. In: *International Conference on Wireless and Mobile Computing, Networking and Communications*. 2015, pp. 217–224. DOI: `10.1109/WiMOB.2015.7347964`.

[AHT16a] W. Ahmad, O. Hasan, and S. Tahar. “Formalization of Reliability Block Diagrams in Higher-order Logic”. In: *Journal of Applied Logic* 18.Supplement C (2016), pp. 19–41. ISSN: 1570-8683. DOI: `https://doi.org/10.1016/j.jal.2016.05.007`.

[AHT16b] W. Ahmed, O. Hasan, and S. Tahar. “Towards Formal Reliability Analysis of Logistics Service Supply Chains using Theorem Proving”. In: *International Workshop on the Implementation of Logics*. Ed. by B. Konev, S. Schulz, and L. Simon. Vol. 40. EPiC Series in Computing. 2016, pp. 1–14. DOI: `10.29007/6177`.

[AM01] Y. Abdeddaïm and O. Maler. “Job-Shop Scheduling Using Timed Automata?” In: *Computer Aided Verification*. Ed. by G. Berry, H. Comon, and A. Finkel. Springer Berlin Heidelberg, 2001, pp. 478–492. ISBN: 978-3-540-44585-2. DOI: `10.1007/3-540-44585-4_46`.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. “Symbolic controller synthesis for discrete and timed systems”. In: *Hybrid Systems II*. Ed. by P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry. Springer Berlin Heidelberg, 1995, pp. 1–20. ISBN: 978-3-540-47519-4. DOI: `10.1007/3-540-60472-3_1`.

[AMY17] J. Abdullah, M. Mohaqeqi, and W. Yi. “Synthesis of Ada code from graph-based task models”. In: *Proc. 32nd ACM Symposium on Applied Computing :* 2017, pp. 1467–1472. ISBN: 978-1-4503-4486-9. DOI: `10.1145/3019612.3019681`.

[Baa+11] S. Baarir, C. Braunstein, E. Encrenaz, J.-M. Ilié, I. Mounier, D. Poitrenaud, and S. Younes. “Feasibility analysis for robustness quantification by symbolic model checking”. In: *Formal Methods in System Design* 39.2 (2011), pp. 165–184. ISSN: 1572-8102. DOI: `10.1007/s10703-011-0121-5`.

[Bai+14] C. Baier, C. Dubslaff, S. Klüppelholz, and L. Leuschner. “Energy-Utility Analysis for Resilient Systems Using Probabilistic Model Checking”. In: *35th Conference on Application and Theory of Petri Nets and Concurrency*. Vol. 8489. LNCS. Invited talk. 2014, pp. 20–39. DOI: `10.1007/978-3-319-07734-5_2`.

[Bai+17] C. Baier, J. Klein, L. Leuschner, D. Parker, and S. Wunderlich. “Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes”. In: *Proc. of the 29th International Conference on Computer Aided Verification, Part I.* Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 160–180. DOI: `10.1007/978-3-319-63387-9_8`.

[Bau+95] R. Baumann, T. Hossain, S. Murata, and H. Kitagawa. “Boron compounds as a dominant source of alpha particles in semiconductor devices”. In: *International Reliability Physics Symposium.* 1995, pp. 297–302. DOI: `10.1109/RELPHY.1995.513695`.

[Bau05] R. Baumann. “Soft errors in advanced computer systems”. In: *Design Test of Computers* 22.3 (2005), pp. 258–266. ISSN: 0740-7475. DOI: `10.1109/MDT.2005.69`.

[BC10] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* 1st. Springer, 2010. ISBN: 3642058809, 9783642058806.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.

[Blo+07] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. “Automatic Hardware Synthesis from Specifications: A Case Study”. In: 2007, pp. 1–6. DOI: `10.1109/DATE.2007.364456`.

[Češ+17] M. Češka, F. Dannenberg, N. Paoletti, M. Kwiatkowska, and L. Brim. “Precise parameter synthesis for stochastic biochemical systems”. In: *Acta Informatica* 54.6 (2017), pp. 589–623. ISSN: 1432-0525. DOI: `10.1007/s00236-016-0265-2`.

[Che+13] T. Chen, T. Han, M. Kwiatkowska, and H. Qu. *Efficient Probabilistic Parameter Synthesis for Adaptive Systems.* Tech. rep. RR-13-04. DCS, 2013, p. 13.

[CJP18] R. Calinescu, K. Johnson, and C. Paterson. “Efficient Parametric Model Checking Using Domain-specific Modelling Patterns”. In: *International Conference on Software Engineering: New Ideas and Emerging Results.* ICSE-NIER '18. ACM, 2018, pp. 61–64. ISBN: 978-1-4503-5662-6. DOI: `10.1145/3183399.3183404`.

[Cla+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. “Counterexample-Guided Abstraction Refinement”. In: *Computer Aided Verification.* Ed. by E. A. Emerson and A. P. Sistla. Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4. DOI: `10.1007/10722167_15`.

[CLM89] E. M. Clarke, D. E. Long, and K. L. McMillan. “Compositional model checking”. In: *Fourth Annual Symposium on Logic in Computer Science.* 1989, pp. 353–362. DOI: `10.1109/LICS.1989.39190`.

[Cub+17] M. Cubuktepe, N. Jansen, S. Junges, J.-P. Katoen, I. Papusha, H. A. Poonawala, and U. Topcu. “Sequential Convex Programming for the Efficient Verification of Parametric MDPs”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by A. Legay and T. Margaria. Springer Berlin Heidelberg, 2017, pp. 133–150. ISBN: 978-3-662-54580-5.

[Deh+17] C. Dehnert, S. Junges, J. Katoen, and M. Volk. “A Storm is Coming: A Modern Probabilistic Model Checker”. In: *Computer Aided Verification.* Vol. 10427. LNCS. 2017, pp. 592–600. DOI: `10.1007/978-3-319-63390-9_31`.

[DF09] R. Dimitrova and B. Finkbeiner. “Synthesis of Fault-Tolerant Distributed Systems”. In: *Automated Technology for Verification and Analysis.* 2009, pp. 321–336. DOI: `10.1007/978-3-642-04761-9_24`.

[Dum+07] E. Dumitrescu, A. Girault, H. Marchand, and E. Rutten. “Optimal discrete controller synthesis for the modeling of fault-tolerant distributed systems”. In: *IFAC Proceedings Volumes* 40.6 (2007). Workshop on Dependable Control of Discrete Systems, pp. 169–174. ISSN: 1474-6670. DOI: `https://doi.org/10.3182/20070613-3-FR-4909.00031`.

[EC80] E. A. Emerson and E. M. Clarke. “Characterizing correctness properties of parallel programs using fixpoints”. In: *Automata, Languages and Programming.* Ed. by J. de Bakker and J. van Leeuwen. Springer Berlin Heidelberg, 1980, pp. 169–181. ISBN: 978-3-540-39346-7. DOI: `10.1007/3-540-10003-2_69`.

[EET13a] EETimes. *Toyota Case: Single Bit Flip That Killed.* 2013. URL: `http://www.eetimes.com/document.asp?doc_id=1319903`.

[EET13b] EETimes. *Transcript of Morning Trial Proceedings.* 2013. URL: `http://www.safetyresearch.net/Library/Bookout_v_Toyota_Barr_REDACTED.pdf`.

[EL16] J. Ezekiel and A. Lomuscio. “Combining fault injection and model checking to verify fault tolerance, recoverability, and diagnosability in multi-agent systems”. In: *Information and Computation* (2016), pp. 167–194. ISSN: 0890-5401. DOI: `http://dx.doi.org/10.1016/j.ic.2016.10.007`.

[ET99] E. A. Emerson and R. J. Trefler. “From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking”. In: *Correct Hardware Design and Verification Methods.* Ed. by L. Pierre and T. Kropf. Springer Berlin Heidelberg, 1999, pp. 142–157. ISBN: 978-3-540-48153-9. DOI: `10.1007/3-540-48153-2_12`.

[FFR01] M. S. Feather, S. Fickas, and N. Razermera-Mamy. “Model-Checking for Validation of a Fault Protection System”. In: *IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, 2001, pp. 32–41.

[Frä+15] M. Fränzle, S. Gerwinn, P. Kröger, A. Abate, and J.-P. Katoen. “Multi-objective Parameter Synthesis in Probabilistic Hybrid Systems”. In: *Formal Modeling and Analysis of Timed Systems*. Ed. by S. Sankaranarayanan and E. Vicario. Springer International Publishing, 2015, pp. 93–107. ISBN: 978-3-319-22975-1. DOI: `10.1007/978-3-319-22975-1_7`.

[FYO14] L. Fang, Y. Yamagata, and Y. Oiwa. “Evaluation of A Resilience Embedded System Using Probabilistic Model-Checking”. In: *International Workshop on Engineering Safety and Security Systems* 150 (2014), pp. 35–49. ISSN: 2075-2180. DOI: `10.4204/eptcs.150.4`.

[Gal15] J. H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving, Second Edition*. Dover Publications, Inc., 2015. ISBN: 0486780821, 9780486780825.

[GL91] O. Grumberg and D. E. Long. “Model checking and modular verification”. In: *International Conference on Concurrency Theory*. Ed. by J. C. M. Baeten and J. F. Groote. Springer Berlin Heidelberg, 1991, pp. 250–265. ISBN: 978-3-540-38357-4. DOI: `10.1007/3-540-54430-5_93`.

[Gme+14] A. Gmeiner, I. Konnov, U. Schmid, H. Veith, and J. Widder. “Tutorial on Parameterized Model Checking of Fault-Tolerant Distributed Algorithms”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Vol. 8483. LNCS. Springer, 2014, pp. 122–171. DOI: `10.1007/978-3-319-07317-0_4`.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Vol. 1032. LNCS. Springer, 1996. ISBN: 9783540607618.

[GR09] A. Girault and É. Rutten. “Automating the addition of fault tolerance with discrete controller synthesis”. In: *Formal Methods in System Design* 35.2 (2009), p. 190. ISSN: 1572-8102. DOI: `10.1007/s10703-009-0084-y`.

[Gun+14] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. “What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC ’14. ACM, 2014, 7:1–7:14. ISBN: 978-1-4503-3252-1. DOI: `10.1145/2670979.2670986`.

[Her+18a] L. Herrmann, C. Baier, C. Fetzer, S. Klüppelholz, and M. Napierkowski. “Formal Parameter Synthesis for Energy-Utility-Optimal Fault Tolerance”. In: *Proc. of the 15th European Performance Engineering Workshop*. Ed. by R. Bakhshi, P. Ballarini, B. Barbot, H. Castel-Taleb, and A. Remke. Lecture Notes in Computer Science. Accepted for publication. Springer, 2018, pp. 78–93. DOI: `10.1007/978-3-030-02227-3_6`.

[Her+18b] L. Herrmann, M. Küttler, T. Stumpf, C. Baier, H. Härtig, and S. Klüppelholz. “Configuration of Inter-Process Communication with Probabilistic Model Checking”. In: *Foundations of Mastering Change* (2018). to appear.

[HKM08] T. Han, J. Katoen, and A. Mereacre. “Approximate Parameter Synthesis for Probabilistic Time-Bounded Reachability”. In: *IEEE Real-Time Systems Symposium*. 2008, pp. 173–182. DOI: `10.1109/RTSS.2008.19`.

[HM18] S. Haddad and B. Monmege. “Interval Iteration Algorithm for MDPs and IMDPs”. In: *Theoretical Computer Science* 735 (2018), pp. 111–131. DOI: `10.1016/j.tcs.2016.12.003`.

[HM80] M. Hennessy and R. Milner. “On observing nondeterminism and concurrency”. In: *Automata, Languages and Programming*. Ed. by J. de Bakker and J. van Leeuwen. Springer Berlin Heidelberg, 1980, pp. 299–309. ISBN: 978-3-540-39346-7.

[Hoq+14] K. A. Hoque, O. Ait Mohamed, Y. Savaria, and C. Thibeault. “Early Analysis of Soft Error Effects for Aerospace Applications Using Probabilistic Model Checking”. In: *Formal Techniques for Safety-Critical Systems*. Ed. by C. Artho and P. C. Ölveczky. Springer International Publishing, 2014, pp. 54–70. ISBN: 978-3-319-05416-2.

[Hsi00] F.-S. Hsieh. “Reconfigurable fault tolerant deadlock avoidance controller synthesis for assembly production processes”. In: *IEEE International Conference on Systems, Man and Cybernetics*. Vol. 4. 2000, 3045–3050 vol.4. DOI: `10.1109/ICSMC.2000.884465`.

[Jak+15] B. M. Jakosky, R. P. Lin, J. M. Grebowsky, J. G. Luhmann, D. F. Mitchell, G. Beutelschies, T. Priser, M. Acuna, L. Andersson, D. Baird, D. Baker, R. Bartlett, M. Benna, S. Bougher, D. Brain, D. Carson, S. Cauffman, P. Chamberlin, J.-Y. Chaufray, O. Cheatom, J. Clarke, J. Connerney, T. Cravens, D. Curtis, G. Delory, S. Demcak, A. DeWolfe, F. Eparvier, R. Ergun, A. Eriksson, J. Espley, X. Fang, D. Folta, J. Fox, C. Gomez-Rosa, S. Habenicht, J. Halekas, G. Holsclaw, M. Houghton, R. Howard, M. Jarosz, N. Jedrich, M. Johnson, W. Kasprzak, M. Kelley, T. King, M. Lankton, D. Larson, F. Leblanc, F. Lefevre, R. Lillis, P. Mahaffy, C. Mazelle, W. McClintock, J. McFadden, D. L. Mitchell, F. Montmessin, J. Morrissey, W. Peterson, W. Possel, J.-A. Sauvaud, N. Schneider, W. Sidney, S. Sparacino, A. I. F. Stewart, R. Tolson, D. Toublanc, C. Waters, T. Woods, R. Yelle, and R. Zurek. “The Mars Atmosphere and Volatile

Evolution (MAVEN) Mission". In: *Space Science Reviews* 195.1 (2015), pp. 3–48.

[Jia+17] X. Jiang, N. Guan, X. Long, and W. Yi. "Semi-Federated Scheduling of Parallel Real-Time Tasks on Multiprocessors". In: *CoRR* abs/1705.03245 (2017). arXiv: `1705.03245`.

[Kim+14] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors". In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture*. 2014, pp. 361–372. DOI: `10.1109/ISCA.2014.6853210`.

[KNP07] M. Kwiatkowska, G. Norman, and D. Parker. "Stochastic Model Checking". In: *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation*. Ed. by M. Bernardo and J. Hillston. Vol. 4486. LNCS. Springer, 2007, pp. 220–270. DOI: `10.1007/978-3-540-72522-0_6`.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Computer Aided Verification*. Vol. 6806. LNCS. 2011, pp. 585–591. DOI: `10.1007/978-3-642-22110-1_47`.

[Knu11] D. E. Knuth. *The Art of Computer Programming: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 2011. ISBN: 0201038048, 9780201038040.

[KPP12] U. Klein, N. Piterman, and A. Pnueli. "Effective Synthesis of Asynchronous Systems from GR(1) Specifications". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by V. Kuncak and A. Rybalchenko. Springer Berlin Heidelberg, 2012, pp. 283–298. ISBN: 978-3-642-27940-9.

[Kul95] V. G. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, Ltd., 1995. ISBN: 0412049910.

[Kuv+16] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer. "HAFT: Hardware-assisted Fault Tolerance". In: *European Conference on Computer Systems (Eurosys)*. ACM, 2016. DOI: `10.1145/2901318.2901339`.

[Kwi+10] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. "Assume-Guarantee Verification for Probabilistic Systems". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by J. Esparza and R. Majumdar. Springer Berlin Heidelberg, 2010, pp. 23–37. ISBN: 978-3-642-12002-2.

[Leu+17] L. Leuschner, M. Küttler, T. Stumpf, C. Baier, H. Härtig, and S. Klüppelholz. “Towards Automated Configuration of Systems with Non-Functional Constraints”. In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS’17. ACM, 2017, pp. 111–117. DOI: `10.1145/3102980.3102999`.

[Li+04] L. Li, V. Degalahal, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. “Soft error and energy consumption interactions: a data cache perspective”. In: *Proceedings of the International Symposium on Low Power Electronics and Design*. 2004, pp. 132–137. DOI: `10.1109/LPE.2004.240852`.

[Li+10] X. Li, M. C. Huang, K. Shen, and L. Chu. “A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility”. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’10. USENIX Association, 2010, pp. 6–6.

[Lin98] J. H. V. Lint. *Introduction to Coding Theory*. 3rd. Springer-Verlag, 1998. ISBN: 3540641335.

[Lip75] R. J. Lipton. “Reduction: A Method of Proving Properties of Parallel Programs”. In: *Communications of the ACM* 18.12 (1975), pp. 717–721. ISSN: 0001-0782. DOI: `10.1145/361227.361234`.

[LR05] E. L. Lehmann and J. P. Romano. *Testing statistical hypotheses*. Springer Texts in Statistics. Springer, 2005, pp. xiv+784. ISBN: 0-387-98864-5.

[LR16] F. Long and M. Rinard. “Automatic Patch Generation by Learning Correct Code”. In: *SIGPLAN Not.* 51.1 (2016), pp. 298–312. ISSN: 0362-1340. DOI: `10.1145/2914770.2837617`.

[Maz87] A. Mazurkiewicz. “Trace theory”. In: *Petri Nets: Applications and Relationships to Other Models of Concurrency*. Ed. by W. Brauer, W. Reisig, and G. Rozenberg. Springer Berlin Heidelberg, 1987, pp. 278–324. ISBN: 978-3-540-47926-0.

[McK+96] W. R. McKee, H. P. McAdams, E. B. Smith, J. W. McPherson, J. W. Janzen, J. C. Ondrusek, A. E. Hyslop, D. E. Russell, R. A. Coy, D. W. Bergman, N. Q. Nguyen, T. J. Aton, L. W. Block, and V. C. Huynh. “Cosmic ray neutron induced upsets as a major contributor to the soft error rate of current and future generation DRAMs”. In: *Proceedings of International Reliability Physics Symposium*. 1996, pp. 1–6. DOI: `10.1109/RELPHY.1996.492052`.

[Meu+17] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3:e103 (2017). ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`.

[MW79] T. C. May and M. H. Woods. “Alpha-particle-induced soft errors in dynamic memories”. In: *IEEE Transactions on Electron Devices* 26.1 (1979), pp. 2–9. ISSN: 0018-9383. DOI: `10.1109/T-ED.1979.19370`.

[NDO11] E. B. Nightingale, J. R. Douceur, and V. Orgovan. “Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs”. In: *Proceedings of the Sixth Conference on Computer Systems*. ACM, 2011, pp. 343–356. ISBN: 978-1-4503-0634-8. DOI: `10.1145/1966445.1966477`.

[Nor+04] G. Norman, D. Parker, M. Z. Kwiatkowska, and S. K. Shukla. “Evaluating the Reliability of Defect-Tolerant Architectures for Nanotechnology with Probabilistic Model Checking”. In: *International Conference on VLSI Design*. IEEE Computer Society, 2004, p. 907. ISBN: 9780470316887.

[Păs+08] C. S. Păsăreanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. “Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning”. In: *Formal Methods in System Design* 32.3 (2008), pp. 175–205. ISSN: 1572-8102. DOI: `10.1007/s10703-008-0049-6`.

[Pel96] D. A. Peled. “Combining Partial Order Reductions with On-the-Fly Model-Checking”. In: *Formal Methods in System Design* 8.1 (1996), pp. 39–64.

[Put94] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 1994.

[Sah05] G. K. Saha. “Approaches to Software Based Fault Tolerance”. In: *Computer Science Journal of Moldova* 13.2 (2005), pp. 193–231.

[Sam16] Samsung. *Samsung Starts Industry’s First Mass Production of System-on-Chip with 10-Nanometer FinFET Technology*. press release. 2016. URL: `https://news.samsung.com/global/samsung-starts-industrys-first-mass-production-of-system-on-chip-with-10-nanometer-finfet-technology`.

[SBS15] H. Schirmeier, C. Borchert, and O. Spinczyk. “Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors”. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 319–330. DOI: `10.1109/DSN.2015.44`.

[Sch+98] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann. “Validating Requirements for Fault Tolerant Systems using Model Checking”. In: *3rd International Conference on Requirements Engineering*. IEEE Computer Society, 1998, pp. 4–13.

[SD15] M. Seaborn and T. Dullien. *Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges*. 2015. URL: `https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`.

[Seg95] R. Segala. "Modeling and Verification of Randomized Distributed Real-Time Systems". PhD thesis. MIT, 1995.

[SG10] B. Schroeder and G. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems". In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (2010), pp. 337–350. ISSN: 1545-5971. DOI: `10.1109/TDSC.2009.4`.

[Sha+14] M. Shafique, S. Garg, J. Henkel, and D. Marculescu. "The EDA challenges in the dark silicon era". In: *51st ACM/EDAC/IEEE Design Automation Conference*. 2014, pp. 1–6. DOI: `10.1145/2593069.2593229`.

[Sha48a] C. E. Shannon. "A Mathematical Theory of Communication". In: *Bell Systems Technical Journal* 27 (1948), pp. 623–656.

[Sha48b] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. ISSN: 0005-8580. DOI: `10.1002/j.1538-7305.1948.tb01338.x`.

[Shi+02] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, I. Technical, C. J. Keaty, R. Bell, and R. Rajamony. "Modeling the effect of technology trends on the soft error rate of combinational logic". In: *Dependable Systems and Networks*. 2002, pp. 389–398. DOI: `10.1109/DSN.2002.1028924`.

[SL12] V. Sridharan and D. Liberty. "A study of DRAM failures in the field". In: *High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: `10.1109/SC.2012.13`.

[Sri+13] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. "Feng Shui of supercomputer memory positional effects in DRAM and SRAM faults". In: *International Conference for High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–11.

[Sup14] W. Suparta. "Space Weather Effects on Microelectronics Devices around the LEO Spacecraft Environments". In: *Journal of Physics: Conference Series* 539.1 (2014), p. 012025.

[TSM18] TSMC. *TSMC Breaks Ground on Fab 18 in Southern Taiwan Science Park*. press release. 2018. URL: `http://www.tsmc.com/tsmcdotcom/PRListingNewsAction.do?action=detail&language=E&newsid=THGOHITHTH`.

[Var85] M. Y. Vardi. "Automatic verification of probabilistic concurrent finite state programs". In: *26th Annual Symposium on Foundations of Computer Sciencea*. 1985, pp. 327–338. DOI: `10.1109/SFCS.1985.12`.

[Wei18] C. Wei. *Q1 2018 Taiwan Semiconductor Manufacturing Co Ltd Earnings Call*. earnings call. 2018. URL: `http://www.tsmc.com/uploadfile/ir/quarterly/2018/1yZjH/E/TSMC%201Q18%20transcript.pdf`.

[Wu+12] Y. Wu, G. Huang, H. Song, and Y. Zhang. “Model Driven Configuration of Fault Tolerance Solutions for Component-Based Software System”. In: *Model Driven Engineering Languages and Systems*. Ed. by R. B. France, J. Kazmeier, R. Breu, and C. Atkinson. Springer Berlin Heidelberg, 2012, pp. 514–530. ISBN: 978-3-642-33666-9.

[WZH07] B. Wachter, L. Zhang, and H. Hermanns. “Probabilistic Model Checking Modulo Theories”. In: *Fourth International Conference on the Quantitative Evaluation of Systems*. 2007, pp. 129–140. DOI: `10.1109/QEST.2007.10`.

[YS02] H. L. S. Younes and R. G. Simmons. “Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling”. In: *Computer Aided Verification*. Ed. by E. Brinksma and K. G. Larsen. Springer Berlin Heidelberg, 2002, pp. 223–235. ISBN: 978-3-540-45657-5.

[YS05] W. L. Yeung and S. A. Schneider. “Formal verification of fault-tolerant software design: the CSP approach”. In: *Microprocessors and Microsystems* 29.5 (2005), pp. 197–209. DOI: `10.1093/comjnl/43.3.191`.

[Zie98] J. F. Ziegler. “Terrestrial cosmic ray intensities”. In: *IBM Journal of Research and Development* 42.1 (1998), pp. 117–140. ISSN: 0018-8646. DOI: `10.1147/rd.421.0117`.