Quantitative Modeling and Verification of Evolving Software

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Sinem Getir Yaman, M.Sc.

Präsidentin der Humboldt-Universität zu Berlin:
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftliche Fakultät:
Prof. Dr. Elmar Kulke

Gutachter/innen:

1. Prof. Dr. Lars Grunske
2. Prof. Dr. Holger Schlingloff
3. Prof. Dr. Matthias Tichy

Tag der mündlichen Prüfung: 25 June 2021

This work is licensed under a Creative Commons License:
Attribution 4.0 International
To view a copy of this license visit
http://creativecommons.org/licenses/by/4.0/

Sinem Getir Yaman: *Quantitative Modeling and Verification of Evolving Software*, DISSERTATION
zur Erlangung des akademischen Grades im Fach Informatik, © December 2020

Published online at the
open-access publication server by the Humboldt-Universität zu Berlin

Dedicated to my parents *Bihter* and *Talih Getir*.

ABSTRACT

Software plays an innovative role in many different domains, such as car industry, autonomous and smart systems, and communication. Hence, the quality of the software is of utmost importance and needs to be properly addressed during software evolution.

Several approaches have been developed to evaluate systems' quality attributes, such as reliability, safety, and performance of software. Models, such as Markov models, fault trees, and Petri nets, commonly serve as models of the software to be built for different evaluation procedures. These models usually contain design-time estimates for specific transition probabilities, which have to be checked at run-time for their accurateness. Furthermore, due to the dynamic nature of modern software systems, these models and their transition probabilities change over time and fluctuate, leading to a significant problem that needs to be solved to obtain correct evaluation results of quantitative properties. Probabilistic models need to be continually updated at run-time to solve this issue. However, continuous re-evaluation of complex probabilistic models is expensive. Recently, incremental approaches have been found to be promising for the verification of evolving and self-adaptive systems. Nevertheless, substantial improvements have not yet been achieved for evaluating structural changes in the model.

Probabilistic systems are usually modeled as automata with probabilistic data (e.g., Markov chains, Markov Decision Processes, probabilistic automata). Even algebraic languages, such as stochastic algebras, are analyzed on the underlying models like Markov chains. Such models can be represented in a matrix form to solve the equations based on states and transition probabilities. On the other side, such changes can create various effects on the models and force one to re-verify the whole system. Run-time models, such as matrices or graph representations, lack the expressiveness to identify the change effect on the model.

In this thesis, we develop a framework using stochastic regular expression trees, which are modular, with action-based probabilistic logic in the model checking context. Such a modular framework enables us to develop change operations for the incremental computation of local changes that can occur in the model. Furthermore, we describe probabilistic change patterns to apply efficient *incremental quantitative verification* using stochastic regular expression trees and evaluate our results.

Keywords: Probabilistic verification, probabilistic model checking, stochastic regular expressions, probabilistic regular expressions, incremental verification, software evolution, software quality, reliability, performance, system safety.

ZUSAMMENFASSUNG

Mit der steigenden Nachfrage nach Innovationen spielt Software in verschiedenen Wirtschaftsbereichen eine wichtige Rolle, wie z.B. in der Automobilindustrie, bei autonomen und intelligenten Systemen als auch bei Kommunikationssystemen. Daher ist die Qualität für die Softwareentwicklung von großer Bedeutung.

Es wurden verschiedene Ansätze entwickelt, um die Qualitätsmerkmale von Systemen wie Zuverlässigkeit, Sicherheit (Safety) und Performance von Software zu bewerten. In der Regel dienen Modelle wie z.B. die der Markov-Modelle, Fehlerbäume und Petri-Netze als Modelle der Software, die für die Erstellung der verschiedenen Auswertungsverfahren ausschlaggebend sind. Normalerweise enthalten diese Modelle Entwurfszeitschätzungen für bestimmte Übergangswahrscheinlichkeiten, die zur Laufzeit auf ihre Genauigkeit überprüft werden müssen. Allerdings ändern sich diese Modelle angesichts der dynamischen Natur moderner Softwaresysteme. Dies führt dazu, dass ihre Übergangswahrscheinlichkeiten im Laufe der Zeit schwanken, welches zu erheblichen Problemen führt. Um korrekte Bewertungsergebnisse quantitativer Eigenschaften erhalten zu können, müssen diese gelöst werden. Dahingehend werden probabilistische Modelle im Hinblick auf ihre Laufzeit kontinuierlich aktualisiert. Eine fortdauernde Neubewertung komplexer Wahrscheinlichkeitsmodelle ist jedoch teuer. In letzter Zeit haben sich inkrementelle Ansätze als vielversprechend für die Verifikation von adaptiven Systemen erwiesen. Trotzdem wurden bei der Bewertung struktureller Änderungen im Modell noch keine wesentlichen Verbesserungen erzielt. Wahrscheinlichkeitssysteme werden als Automaten mit Wahrscheinlichkeitsdaten modelliert, wie bei Markov-Modellen, Markov-Entscheidungsprozessen und probabilistischen Automaten. Auch werden algebraische Sprachen wie stochastische Algebren wie Differentialgleichungen auf Basis von Markov-Ketten analysiert. Solche Modelle können in Matrixform dargestellt werden, um die Gleichungen basierend auf Zuständen und Übergangswahrscheinlichkeiten zu lösen.

Auf der anderen Seite können solche Änderungen auch verschiedene Auswirkungen auf die Modelle haben oder eine erneute Überprüfung des gesamten Systemmodells erzwingen. Laufzeitmodelle wie Matrizen oder Diagrammdarstellungen sind nicht signifikant, um die Auswirkungen von Modellveränderungen erkennen zu können.

In dieser Arbeit wird ein Framework unter Verwendung stochastischer Bäume mit regulären Ausdrücken entwickelt, welches modular aufgebaut ist und eine aktionshaltige sowie probabilistische Logik im Kontext der Modellprüfung aufweist. Ein solches modulares Framework ermöglicht dem Menschen die Entwicklung der Änderungsoperationen für die inkrementelle Berechnung lokaler Änderungen, die im Modell auftreten können. Darüber hinaus werden probabilistische Änderungsmuster beschrieben, um eine effiziente inkrementelle Verifizierung, unter Verwendung von Bäumen mit regulären Ausdrücken, anwenden zu können. Durch die Bewertung der Ergebnisse wird der Vorgang abgeschlossen.

In memory of all those who lost their lives and homes,
during Izmir earthquake

— on 30th of October, 2020.

ACKNOWLEDGMENTS

My deepest thanks go to my love *Burak Yaman* for his endless patience and understanding during my up and downs, making my life joyful. This thesis would not successfully finish without his great encouragement for me to progress.

I owe my greatest thanks to my advisor Lars Grunske, who believes and gives me the chance to pursue my research and dissertation. I would appreciate the facilities he provided for me to collaborate with other researchers, which I learned a lot in the field of Software Engineering. I would also like to thank Matthias Tichy for our scientific discussions and sharing his research experiences with me also reviewing my dissertation. My sincere thanks go to my Holger Schlingloff for his valuable thoughts and reviewing my thesis.

I present my deepest gratitude to Esteban Pavese who guides and enlightens me in formal methods and provides myself to see the end of tunnel in my way of dissertation. I am thankful to Timo Kehrer for our discussions not only learning me a lot but also having some fun during difficult times. I am very glad to meet him on my way of research for his great patience and views during our collaborations that has improved my vision. My sincere thanks go to Antonio Filieri for the initial discussions about the incremental verification and research methodologies.

Thanks to my students Jonas Heinisch, Duch Ahn Vu, Tobias Beeh, Robert Duda, Luca Beurer-Kellner, Robert Golda and Lorenz Claus that work with me in various projects and also Lukas Schramm for proofreading my thesis.

I am also grateful to Software Engineering chair members Thomas Vogel, Yannic Noller, Simon Heiden and Mingxing Tang, Sebastian Müller Birgit Schiefner for their companion and help, Birgit Heene for her practical solutions for us making life easier. Many thanks to Kim, Irem, Bo, Samira, Marvin, Matthias, for making my time fun during our lunch and coffee breaks.

I owe to a huge thank to my family that shows a great patience and always encourage me; my hero Talih Getir and my angel Bihter Getir, my lovely sister Sıla for me helping during editing and their great support to her and her husband. A huge thanks goes to friends in Berlin and Stuttgart especially Şengül and Rachel for their companion during my research.

Last but not least, I am also thankful to DFG (German Research Foundation) for funding my research under the priority program*Design for Future*, together with SPP project members for our valuable collaborations, and all people I have met during the conferences, events for their precious insights and recommendations during my PhD journey.

CONTENTS

I PROLOGUE 1

1 INTRODUCTION 3
- 1.1 Quantitative Verification . . . 3
- 1.2 Evolution of QoS (Quantitative) Models . . . 4
- 1.3 Motivation: Incremental Quantitative Verification . . . 5
- 1.4 Roadmap . . . 8

2 BACKGROUND 11
- 2.1 Notes on Regular Languages . . . 11
  - 2.1.1 Linear Equations Solving: From FA to Regex . . . 14
  - 2.1.2 Parsing Regex . . . 16
  - 2.1.3 Constructing FA from Regex . . . 17
  - 2.1.4 Pattern Matching of Regex Using NFA . . . 19
  - 2.1.5 Model Checking Automata of Linear Properties . . . 21
  - 2.1.6 Action-based Computation Tree Logic (ACTL*) . . . 22
- 2.2 Notes on Probability Theory and Quantitative Verification . . . 23
- 2.3 Conventional Probabilistic Model Checking . . . 26
  - 2.3.1 Reachability Probabilities . . . 28
  - 2.3.2 Probabilistic Computation Tree Logic . . . 29
- 2.4 Summary . . . 30

3 RELATED WORKS 31
- 3.1 Problem-related Approaches: Supporting Incremental Quantitative Verification . . . 31
  - 3.1.1 Abstraction Refinement Based Approaches . . . 31
  - 3.1.2 Compositional Verification in Probabilistic Models . . . 32
  - 3.1.3 Parametric Model Checking . . . 32
  - 3.1.4 Incremental Probabilistic Verification . . . 33
  - 3.1.5 Configuration Based Approaches . . . 36
- 3.2 Language Related Approaches: Probabilistic Algebraic Expressions . . . 36

II QUANTITATIVE VERIFICATION OF STOCHASTIC REGULAR EXPRESSIONS 39

4 A NEW FORMALISM FOR QUANTITATIVE VERIFICATION 41
- 4.1 Formal Semantics for Probabilistic Verification of SREs . . . 41
- 4.2 Model Checking Using Pattern Matching . . . 47
  - 4.2.1 Translating PACTL Word Formulas into SREs . . . 47
  - 4.2.2 Reachability Analysis . . . 52
  - 4.2.3 Generalized Algorithm . . . 58
- 4.3 Illustrative Example . . . 58
- 4.4 Preliminary Evaluation . . . 63
- 4.5 Discussion . . . 64

5 EQUIVALENCE BETWEEN PROBABILISTIC RABIN AUTOMATA AND STOCHASTIC REGULAR EXPRESSIONS 65
- 5.1 From a PRA to an SRE . . . 65
  - 5.1.1 Conversion Using State Elimination . . . 65

5.1.2 Conversion Using Brzozowski's Algebraic Method: Regular Expression Equation Systems Solving . . . 67
5.1.3 Probabilistic Brzozowski's Algebraic Method . . . 68
5.1.4 Illustrative Example: From a PRA to an SRE . . . 72
5.2 From an SRE to a PRA . . . 74
5.2.1 Glushkov's Construction with Probabilistic Extension . . . 76
5.2.2 Illustrative Example: From an SRE to a PA . . . 82
5.3 Conclusion . . . 84

III INCREMENTAL QUANTITATIVE VERIFICATION 87
6 INCREMENTAL QUANTITATIVE VERIFICATION 89
6.1 Motivation: Using Stochastic Regular Expression Formalism for Incremental Verification . . . 89
6.2 Incremental Quantitative Verification (IQon) Framework . . . 90
6.2.1 SRE Edit Operations for Incremental Analysis . . . 92
6.2.2 Supporting Incremental Verification: Change Patterns in Probabilistic Specification . . . 99
6.3 Preliminary Evaluation: Benefits of Domain Specific Edit Operations for Changing Models . . . 107
6.3.1 The SiDECAR Framework . . . 107
6.3.2 The SRE@SiDECAR Plugin . . . 108
6.4 Incremental Transformations between PRA and SRE Models . . . 112
6.4.1 Incremental Transformations from PRA to SREs . . . 112
6.4.2 Incremental Transformations from SREs to PRA . . . 113
6.5 Conclusion . . . 115

IV VALIDATION 117
7 EVALUATION 119
7.1 Obtaining SRE Models . . . 119
7.1.1 Learning SRE Models . . . 120
7.1.2 Performance Evaluation of Learning SREs . . . 122
7.1.3 Consistency Checking of Probabilistic Models and Software (The *ConsistAnts* Framework) . . . 124
7.1.4 Using *ConsistAnts* for the Consistency Evaluation of pBLA . . . 127
7.1.5 Borg Case Study . . . 131
7.1.6 Transformations from PRA to SRE . . . 133
7.1.7 Application of State Elimination Using Model Transformations . . . 134
7.1.8 Application of Brozowoski Method . . . 135
7.1.9 Comparing Verification Results with PRISM: Sanity Check . . . 136
7.2 Incremental versus Non-Incremental Approach . . . 137
7.2.1 Internal Evaluation . . . 137
7.2.2 External Evaluation . . . 140
7.3 Conclusion . . . 144

V CONCLUSION 147
8 CONCLUSION AND FUTURE WORK 149

BIBLIOGRAPHY 151

LIST OF FIGURES

Figure 1 Quantitative verification of evolving software 5
Figure 2 Overview of probabilistic verification 8
Figure 3 An example FA and accepting runs over words 12
Figure 4 NFA-DFA translation * 13
Figure 5 Changed FA from Figure 3 15
Figure 6 Abstract syntax tree for the regex $(a : b)^* : a : b : a + b : b : a$ . . . 17
Figure 7 FSM constructed from the regex $E = (AT|GA)((AG|AAA)^*)$ . . . 20
Figure 8 Example of pattern matching 20
Figure 9 Automata-based LTL model checking in a nutshell with an example [22] 22
Figure 10 An example PRA 26
Figure 11 Markov chain representing a communication protocol [9] 29
Figure 12 State elimination of parametric Markov chain [36] 33
Figure 13 The Zeroconf protocol for $n = k(a)$ and $n = k + 1$ (b) [53] 34
Figure 14 Parsing nested PACTL formula 49
Figure 15 An example of an SRE model defining a simple message protocol 60
Figure 16 Random SREs up to 700,000 length for the size of the string sought from 1 up to 10. 64
Figure 17 PRA-SRE transformations overview 65
Figure 18 An example of state elimination with probabilities 67
Figure 19 Example PRA representing a simple protocol 72
Figure 20 The FSM constructed from regex $E = b(ab)^* + c$ 75
Figure 21 The PRA constructed from SRE $E = b(ab)^{*0.3}_{[4]} + c_{[6]}$ 77
Figure 22 Partially constructed PRA of example 19: $Snapshot_1$ 83
Figure 23 The final constructed PRA of example 19: $Snapshot_2$ 84
Figure 24 Changing probabilistic model 90
Figure 25 *IQon* framework 91
Figure 26 An illustration of a simple delete operation for the node E_5 . . . 95
Figure 27 Changing SRE model 96
Figure 28 Changing SRE tree 99
Figure 29 Plugin components of arithmetic operations in SiDECAR [16] . . 108
Figure 30 An OPG of SREs as a SiDECAR component 109
Figure 31 *IQon*'s working principle versus Generic SiDECAR methodology 110
Figure 32 Comparison of *IQon* and SRE@SiDECAR for the same changes . 111
Figure 33 Example probabilistic Rabin automata with equations 112
Figure 34 Experimental setup 119
Figure 35 Performance evaluation in time for the pBLA learning method . 123
Figure 36 Architecture of the *ConsistAnts* Tool 125
Figure 37 A trace dataset for the alphabet $\{A, B, C\}$ and its footprint matrix 126
Figure 38 Computation of the automata footprint matrix 127
Figure 39 Execution time (seconds) of *ConsistAnts* for the random data 128
Figure 40 Consistency evaluation 129

Figure 41 Average consistency difference between pBLA and AAlergia approaches in percentage . . . 130
Figure 42 Model complexity comparison between the learned PRA model and the translated PRA model from the learned SRE model . . . 130
Figure 43 Lifecyle of the jobs and the tasks in Borg case study [141] 131
Figure 44 Execution time of *ConsistAnts* for the Borg case study 132
Figure 45 Metamodel of SRE languge and PRA using Eclipse Modelling Framework (EMF [42]) . . . 135
Figure 46 $RA_1.a$) Random experiments on various data sets 139
Figure 47 $RA_1.b$) Leader election case study demonstrating execution time in milliseconds for 40 evolution steps and application of 5 edit operations for each evolution step. . . . 139
Figure 48 RA_2) Comparison of the incremental and non-incremental quantitative verification based on the change size 140
Figure 49 *Changing* DTMC model of TAS 142
Figure 50 1. Labeling of automata for module-state relationship. 2. Conversion of DTMC explicit model (state labeled) to a PRA (transition labeled) . . . 142
Figure 51 The experimental setup for the comparison of evolving models and model checking algorithms . . . 143

LIST OF TABLES

Table 1 Classification of related quantitative verification approaches . . . 35
Table 2 Kleene-like probabilistic languages in the context of SREs 37
Table 3 Summary of definitions 58
Table 4 An example of Glushkov sets 75
Table 5 An attribute schema of the SRE plugin on SiDECAR 109
Table 6 Simple sample set on the alphabet $\Sigma = \{a, b, c\}$ 120
Table 7 Block-wise grouped sample set on the alphabet $\Sigma = \{a, b, c\}$ . . . 120
Table 8 Alignment of the grouped elements 121
Table 9 Model parameters 123
Table 10 Consistency results (%) for the Google Cluster data set 132
Table 11 Transformation measurements for JFLAP. Sizes of input models are indicated by the number of states and the transitions; execution times are reported in seconds. . . . 134
Table 12 Transformation measurements for JFLAP and Henshin. Sizes of input models are indicated by the number of states, execution times are reported in seconds [134] 136
Table 13 Transformation measurements for the leader election case study with Brozowoski algebraic method 136

Table 14 Size and time measurements (ms) on the leader election case study models for the property $\mathcal{P}_{[1,1]}(\mathtt{True}\,\mathcal{U}\,(\mathcal{X}_{elected}\mathtt{True}))$ (Reach., Const. and MC stand for Reachability, Construction and Model Checking respectively). 137
Table 15 Bounded reachability analysis results on leader election case study models for the property $\mathcal{P}_{[0.8,1]}(\mathtt{True}\,\mathcal{U}^{\leqslant 10}\,(\mathcal{X}_{elected}\mathtt{True}))$. 138
Table 16 RA_3: External evaluation for changing models 144

ACRONYMS

ACTL	**A**ction **C**omputation **T**ree **L**ogic
BLA	**B**lockwise **L**eft **A**lignment
BSCC	**B**ottom **S**trongly **C**onnected **C**omponent
CTMC	**C**ontinues-**T**ime **M**arkov **C**hain
DTMC	**D**iscrete-**T**ime **M**arkov **C**hain
EMF	**E**clipse **M**odeling **F**ramework
FA	**F**inite **A**utomata
FPM	**F**ailure **P**ropagation **M**odel
FSM	**F**inite **S**tate **M**achine
FT	**F**ault **T**ree
IQon	**I**ncremental **Q**uantitative Verificati**on**
LQN	**L**ayered **Q**ueueing **N**etwork
LTS	**L**abeled **T**ransition **S**ystem
MDP	**M**arkov **D**ecision **P**rocess
MDE	**M**odel-**D**riven **E**ngineering
NFA	**N**on-Deterministic **F**inite **A**utomata
OPG	**O**perator **P**recedence **G**rammar
PACTL	**P**robabilistic **A**ction **C**omputation **T**ree **L**ogic
pBLA	**P**robabilistic **B**lockwise **L**eft **A**lignment
PCTL	**P**robabilistic **C**omputation **T**ree **L**ogic
PMC	**P**arametric **M**odel **C**hecking
PPU	**P**ick and **P**lace **U**nit
PRA	**P**robabilistic **R**abin **A**utomata
Regex	**Reg**ular **Ex**pression
SCC	**S**trongly **C**onnected **C**omponent
SRE	**S**tochastic **R**egular **E**xpression
TAS	**T**ele **A**sistance **S**ystem
QoS	**Q**uality **o**f **S**ervice

Part I

PROLOGUE

1 INTRODUCTION

1.1 QUANTITATIVE VERIFICATION

The correctness of computerized systems, such as autonomous and smart systems, or communication systems is of vital importance and needs to be properly addressed, especially in safety-critical environments, such as embedded systems in the automotive or aircraft industry where human life is *also* in account.

Thus, a proper operation of these systems is increasingly demanded, even though they often operate in unpredictable or unreliable environments. Therefore, designers need other ways to model systems with uncertainty and guarantee that the systems will operate at a certain **quality of service (QoS)** level. Some QoS attributes are reliability, performance (in business information systems), and safety (in embedded systems). In other words, deriving guarantees on precisely specified levels of quality (performance or efficiency) is a valuable method in the design of such systems.

> **Quantitative verification** is a technique for analyzing quantitative aspects of a system's design, such as reliability or performance [120].

It applies formal methods by analyzing a mathematical model to automatically prove certain specified quality properties for **QoS Evaluation** (Figure 1, arrows 5 and 6). Examples of such properties:

- "The probability that the battery power will not drop under 20 % is greater than 0.99"
- "The probability of both sensors failing simultaneously is less than 0.0001"

The facility to obtain formal guarantees for such properties can have many benefits across a wide range of application domains. For example, in safety-critical systems, it is crucial to obtain some bounds with the probability of certain failures or combinations of these failures, or very important to comply with some constraints on timing and performance.

Formalism for Quality of Service Evaluation

Probabilistic models for quantitative verification are usually constructed from architectural and behavioral models of a system together with usage profile, namely probabilistic data at design time (Figure 1, arrows 2 and 3 respectively).

The generated probabilistic models are usually different for each quality attribute. As an example, discrete-time Markov chains (DTMCs) are used for reliability evaluation [68], while fault trees (FTs) and failure propagation models (FPMs) are commonly used for evaluating system safety [73, 108, 122]. On the other hand, layered queueing networks (LQN) and continuous-time Markov chains (CTMCs) are used to predict performance attributes [10, 11]. A quality evaluation model can be generated directly

or via an intermediate language that is used to construct a variety of possible quality evaluation models. Examples of direct approaches are HiP-HOPS [121, 122], which is used in the safety domain to construct fault trees from annotated Matlab Simulink models, or the approach by Rodrigues et al. [127], which constructs a DTMC from annotated scenario specifications. Examples for intermediate models that are used as a basis to construct a plethora of quality evaluation models are KLAPER (Kernel LAnguage for PErformance and Reliability analysis) [28], the CSM (Core Scenario Model) of the PUMA (Performance by Unified Model Analysis) approach [147], or the annotated Palladio component model [12]. Additionally, the standardized UML MARTE profile can be seen as an intermediate language that is currently used to derive performance, reliability, and availability for the related quality evaluation models [13, 139].

Some techniques to generate models are developed in this domain to learn probabilities from running systems by monitoring techniques [43, 49]. However, these studies are limited to only learning probabilities for developed models.

1.2 EVOLUTION OF QOS (QUANTITATIVE) MODELS

While model-based QoS evaluation approaches have gained significant practical application in reasoning about design alternatives in the early phases of software development and provide the foundation for quality assurance, these approaches have also a couple of limitations. First, most approaches are currently used for design-time reasoning and often assume a static system that does not change at run-time. Second, the constructed quality evaluation models are based on assumptions and estimates about the system at run-time. Examples for such estimates are system call/activation rates which describe the operational profile (Figure 1, arrow 2). If the assumptions do not hold or the estimates are not accurate, the quality evaluation model will not represent the system; thus, the results of the quality evaluations have to be handled rigorously. Third, if the behavior of the system changes, the structure of the system model changes as well. Hence, a proper update in the structure of the evaluation model, and a run-time (time- and resource-efficient) evaluation technique that analyzes it should be available.

Therefore, once system models or the usage profile have evolved, probabilistic models need to be updated correspondingly. Such an incremental update is not a straightforward task as shown in previous papers [59]. For example, the failures of an architectural component must be adequately considered in an associated fault tree model for a system. While this quality requirement can reasonably satisfy a particular snapshot of the system, quality evaluation models typically become outdated when the system evolves, i.e., quality evaluation models and system models can evolve in an inconsistent way. As a consequence, quality evaluation leads to unexpected and highly improper results. An example in the context of hazard analysis of component-based embedded systems is the addition of a new port for a sensor of a component without a corresponding addition of the sensor failures in the relevant fault trees. It will clearly lead to wrong hazard analysis results.

In model-driven engineering (MDE), this problem is investigated under the topics of co-evolution, model synchronization, etc. However, quality evaluation models cannot be fully generated from system models, and most relations between the elements of different models are not simple one-to-one correspondences. In other words, co-evolution cannot be fully automated [59]. At best, developers may be supported with recommending possible co-evolutions, e.g., as in the model-based (co-)evolution framework

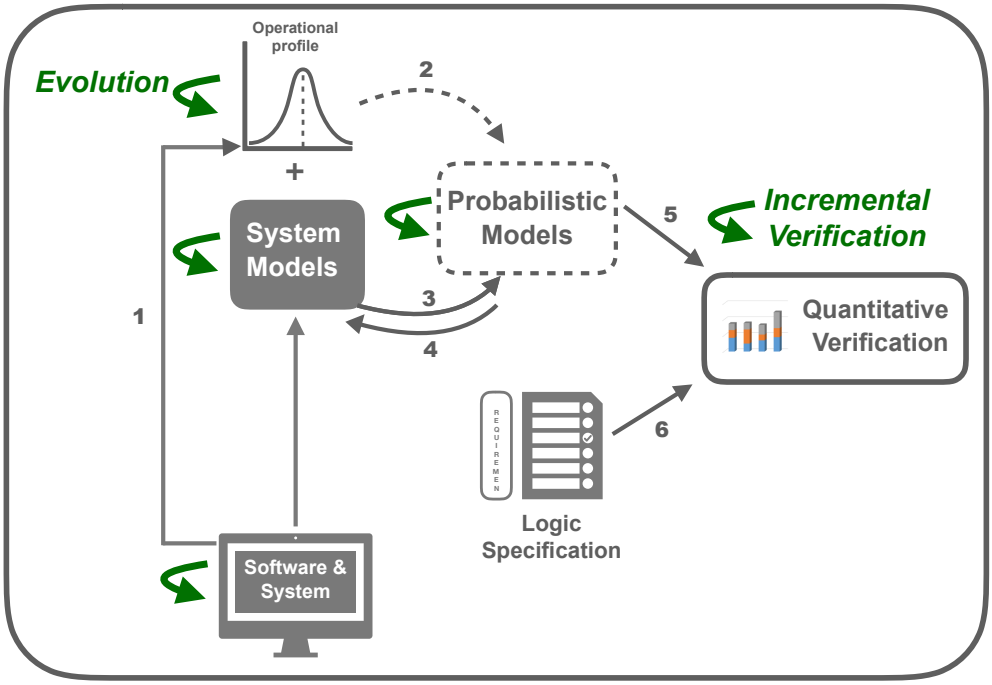


Figure 1: Quantitative verification of evolving software

called CoWolf [62]. To achieve a consistent co-evolution, CoWolf follows a rule-based approach where incremental model transformations are used to evolve one model and co-evolve another model. However, the evolution problem is shifted to the engineering of proper transformation rules since the adequacy of the recommendations strongly depends on the transformation rules being used by the tool. These should capture evolution and co-evolution steps being considered useful by the developers using the tool.

We tackle this problem of proper transformation rule sets in the context of model-based hazard analysis of software-intensive systems, namely system architecture models and fault tree models on the evolution of the so-called Pick and Place Unit (PPU) [105], a case study from the automation engineering domain which is commonly used in the German priority program "Design for Future – Managed Software Evolution" [126]. To study co-evolution in terms of the PPU, we have created consistent software architecture and fault tree models for all safety-relevant evolution scenarios and released the data [57, 58]. Similarly, we study the transformations from state machines to Markov chains and from fault trees to Markov chains in the generic framework CoWolf. We investigate incremental updates among the related models and developed a generic framework for the community [61, 62].

1.3 MOTIVATION: INCREMENTAL QUANTITATIVE VERIFICATION

Dynamically evolving systems have a significant place in the *Horizon 2020* under the scope of Future and Emerging Technologies (self-deployment, self-awareness, adaptation, and evolution) with a €95.50 million budget [2]. To enable quality of service evaluations for dynamically evolving systems, models are required to check at run-time

after any change has occurred to ensure whether the current version satisfies their QoS requirements.

Currently, the quality is usually analyzed at design time under non-perfect knowledge about the behavior of the system and its environment, which can result in incorrect analysis results. Several approaches have been developed to evaluate systems' reliability, safety, and performance properties at design time [10, 68, 72].

A few approaches have been developed to use and update quality evaluation models during system execution [43, 48, 149]. These approaches focus on specific quality evaluation models, such as DTMCs and LQNs, only allow updates of model-specific parameters, like transition probabilities, without considering structural changes of the quality evaluation models. The research on requirements@run has produced significant contributions that provide languages and metamodels to formalize quality requirements [14, 39, 130, 146] as well. These approaches are currently being used to monitor quality requirements directly.

An approach that provides a foundation to apply model-based software engineering techniques for quality of service evaluations at run-time has been developed in [23]. This approach automatically constructs a quality evaluation model and uses a Bayesian update mechanism to align the parameters of the quality evaluation model with the actual system. A complete re-evaluation of a quality evaluation for a complex system is a time-consuming task. An algebraic method [116] for discrete-time Markov chains has been developed to reduce the effort for re-evaluating a model only if a few parameters are updated.

Models, such as Markov models, queuing networks, and Petri nets, serve as inputs for different evaluation procedures. These models usually contain design-time estimates for specific transition probabilities, which have to be checked at run-time for their accurateness. Furthermore, due to the dynamic nature of modern software systems [27], these models and their transition probabilities change over time and fluctuate, leading to a significant challenge about obtaining correct evaluation results of quantitative properties. Therefore, probabilistic models need to be continually updated at run-time [23]; thus, probabilistic evaluation follows the models@runtime [43] philosophy. However, continuous re-evaluation of complex probabilistic models is expensive, especially at run-time. Furthermore, changes usually affect only a small part of the model. Efficient techniques are needed to evaluate the model incrementally during the changes.

Incremental verification is found to be one of the approaches that aims to reduce the overhead of re-verifying the complete system [63].

This thesis serves this purpose of *incremental quantitative verification* by a novel mathematical framework in the scope probabilistic model checking [91], which is a well-established quantitative verification technique developed in order to automatically perform QoS assessments. In the following, we present the contributions of this thesis in detail.

Contribution of the Thesis

I. A novel formalism for incremental verification: A generic mathematical framework that enables the structural changes in the probabilistic model.

Probabilistic systems are usually modeled as automata with probabilistic data (e.g., Markov chains, Markov decision processes, and probabilistic automata). Even algebraic languages like, stochastic algebras, are analyzed on the underlying models such as Markov chains. Such models can be represented in matrix form to solve the equations based on states and transition probabilities [9].

On the other side, changing these models can create substantial effects on the models and leads to re-verify the whole system. The reason why run-time models, such as matrices or graph representations, lack the expressiveness to identify the change effect on the model is that they define edit operations or changing operations that consist of only updating states and transitions. However, adding or deleting a state might affect the model differently. In this regard, we develop a framework using the stochastic regular expression trees, which are modular, with action-based probabilistic logic in the model checking context and used stochastic regular expressions as an input model. Our initial attempt at reachability analysis with strings is promising and convenient for parallel and incremental computation, especially in the domain of component-based systems or modular systems.

Hence, this thesis establishes a new framework for incremental probabilistic model checking by using stochastic regular expression trees that is a different modeling language comparing to the conventional probabilistic model checking. We clarify the scope of the thesis and assumptions in the following paragraphs.

Quantitative and stochastic behaviors of regular expressions have been introduced in several studies such as probabilistic concurrent Kleene algebra in [96]. In this thesis, however, we focus on stochastic regular expressions (SREs) in the model checking context and establish a semantics of SREs employing a probabilistic extension of Action-based Computation Tree Logic (ACTL*) to reason about temporal properties quantitatively. Then, the model checking algorithm is described on the top of the semantics described in some part of our work [60].

In contrast to state-based representations in conventional probabilistic model checking, we introduce an approach and focus on stochastic regular expressions as an input model for probabilistic model checking applications, as demonstrated in Figure 2. SREs are constructed as a tree and provide to localize probabilistic data inside the tree nodes. Thus, such decomposition of probabilistic data can be used for multiple problems such as component-based and iterative verification by enabling the reusability of the calculations for changing probabilistic systems [63].

II. Incremental Quantitative Verification (IQon) framework describing edit operations and change patterns for evolving probabilistic systems.

In addition to the formal foundations of an SRE model checking framework, we have developed edit operations to analyze SRE models for the incremental computation of local changes that can occur in the model. Furthermore, we have described probabilis-

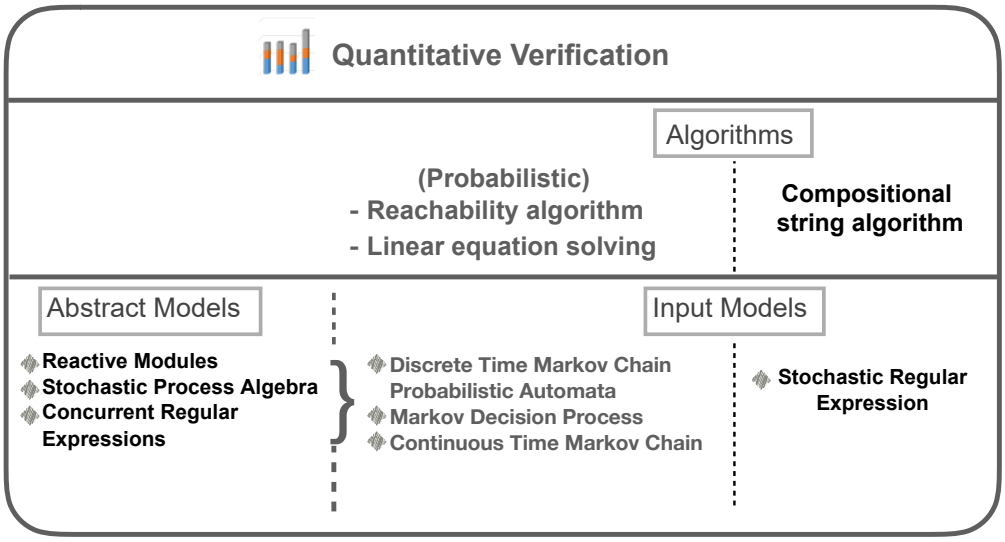


Figure 2: Overview of probabilistic verification

tic change patterns (complex edit operations) to apply *efficient incremental quantitative verification* using stochastic regular expression trees and evaluated our results.

III. Open-source implementation of a probabilistic incremental verification tool, and versions of the case study models in the domain of evolving systems [88].

1.4 ROADMAP

Chapter 2 presents the required formalism and mathematical background. We summarize the basic concepts about regular languages and probabilistic model checking. Furthermore, the fundamental theorems that are used and extended in this thesis are explained.

Then, we review the related studies in the scope of *incremental probabilistic model checking* and *probabilistic algebras* in **Chapter 3**.

Chapter 4 introduces a new formalism for probabilistic model checking of changing models. In contrast to these state-based representations, we introduce an approach and focus on stochastic regular expressions as an input model for probabilistic model checking applications. Stochastic regular expressions are constructed as a tree and provide localization of probabilistic data inside the tree nodes.

An equivalence relationship between stochastic regular expression and probabilistic Rabin automata is provided in **Chapter 5**.

Following the formal foundations, the incremental quantitative verification (IQon) framework is introduced in **Chapter 6** that encapsulates the edit operations to analyze the SRE model for the incremental computation of local changes that can occur in the model. Furthermore, we have described probabilistic change patterns (complex edit operations) to apply efficient incremental verification using regular expression trees and evaluated our results.

Chapter 7 shows the applicability of the proposed approach. We first address how to obtain models and then, apply internal and external evaluation on different data

sets and models including the models of the case study called Tele Assistance System (TAS) from self-adaptive software domain [1]. We also discuss the challenges during the evaluation and the comparison.

Finally, **Chapter 8** summarizes the research problems and solution, presents the future work, and opens questions for the community.

2 BACKGROUND

This section provides a mathematical background for a better understanding of the approaches presented in this thesis. Formal verification by model checking SREs firstly requires the understanding of model checking finite automata and regular expressions. The reason for that is the semantics of SREs over sequences are constructed on traces and prefixes of the language.

We first provide basics on regular languages and the relationship (conversions) between labeled transition systems and regular expressions. Subsequently, an overview of probability theory, randomness, and discrete event systems as well as probabilistic model checking are summarized.

2.1 NOTES ON REGULAR LANGUAGES

In this section, we provide some notes on the basic automata theory [32] to keep a consistency of the definitions throughout the thesis.

Definition 2.1 (Finite Automata)
A finite automaton (FA) is a mathematical model of a device that has a constant amount of memory, independent of the size of its input [32]. Formally, FA[1] *is a tuple* $A = (Q, \Sigma, \delta, Q^0, F)$ *where*

- Q *is a countable set of states,*
- Σ *is an alphabet of visible actions, and the empty transition is denoted by* $\epsilon \notin \Sigma$*,*
- δ *is a transition function such that* $\delta : Q \times (\Sigma \cup \epsilon) \to Q$*,*
- $Q^0 \in Q$ *is a set of initial states, and*
- $F \subseteq Q$ *is a set of final states.*

Definition 2.2 (A Run (ρ))
Let v *be a word (string, action sequence) of* Σ^* *whose length is denoted as* $|v|$*. A run over a word* $v \in \Sigma^*$ *is a mapping such that:*

- *The first state is an initial state,* $\rho(0) \in Q^0$*.*
- *The states are relevant to the transition relation, meaning that, moving from the* i*th state* $\rho(i)$ *to the* $(i+1)$*th state* $\rho(i+1)$ *by reading the* i*th input letter* $v(i)$ *is related to the transition relation* $(\rho(i), v(i), \rho(i+1)) \in \delta$ *for* $0 \leqslant i < |v|$*.*

A *run* is a path in an automation A from q_0 to a state $\rho(|v|)$ where the edges are labeled with letters in v. A run ρ over v is an accepting run if it ends in an accepting state, that is, $\rho(|v|) \in F$. A accepts a word v iff there exists an accepting run of A on v. The language of A, $L(A) \subseteq \Sigma^*$ consists of all the words accepted by A [32].

[1] FA is used interchangeably with finite state machine (FSM) throughout the thesis.

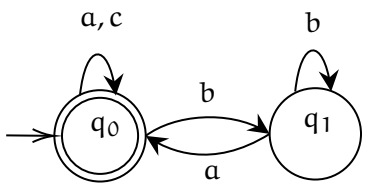


Figure 3: An example FA and accepting runs over words

Example 2.1 (An example of a FA and its *runs*)
A simple example of FA is provided in Figure 3 and some of its ***runs*** *are listed as follows:*

- *A run through the states* q_0, q_1, q_1, q_1, q_0 *on word* $bbba$ *is accepting.*
- *A run through the states* q_0, q_1, q_1, q_0, q_0 *on word* $bbac$ *is accepting.*
- *A run through the states* q_0, q_0, q_1, q_1, q_1 *on word* $cbbb$ *is rejecting.*
- *The language* $L(A)$ *is consisted of* $\{\varepsilon, a, ac, abaa, \ldots\}$.
- *The regular expression of* $L(A)$ *is* $(a + c + b : b^* : a)^*$.

An automaton is *deterministic* iff the transition relation (q, ϵ, q') implies $q = q'$, and the transition relations (q, ϵ, q') and (q, ϵ, q'') imply $q' = q''$ for all q, q', and q''. Otherwise, it is non-deterministic because there exist multiple choices between multiple state candidates during performing a state transition. Therefore, the execution runs in parallel to try possible candidates that can lead to the acceptance state [32].

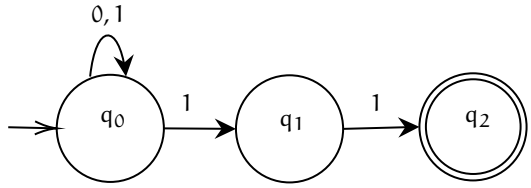
Every non-deterministic finite automaton (NFA) can be translated into a language-equivalent deterministic automaton (DFA) (an example in Figure 4). For any two states q and q', and any action sequence $a_1, \cdots a_n \in \Sigma \cup \epsilon$, q' is *reachable* from q if there exists a sequence $q \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'$.

Example 2.2 (Equivalent NFA-DFA)
In figure 4, we present a NFA (Figure 4a) and a converted DFA (Figure 4b) by eliminating the non-determinism in the given NFA (e.g., adding the state q_3 *to remove the choice from* q_1 *that is the transition* 0 *to multiple states:* $(q_1, 0, q_2)$ *and* $(q_1, 0, q_1)$*).*
As an instance of reachability, q_3 *is reachable from* q_0*, whereas* q_1 *is not reachable from* q_2 *in Figure 4b.*

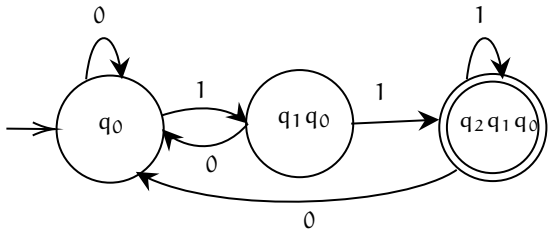
Definition 2.3 (Infinite run and Büchi automata)
The simplest automata over infinite words are Büchi automata. A Büchi automaton has the same structure as FA over finite words, but different notion of acceptance, meaning that F *is called accepting states rather than final states. A run* ρ *of a Büchi automaton is defined over an infinite word* $v \in \Sigma^\omega$.
Let $\inf(\rho)$ *be the set of states that appears infinitely often in* ρ*:*

$$\inf(\rho) = \{q | \rho(j) = q, \forall i \in \mathcal{N} \text{ and } \exists j \geqslant i\}.$$

Consequently, a run ρ is accepting (Büchi accepting) iff $\inf(\rho) \cap F \neq \emptyset$, and a set of strings is ω-regular iff it is recognizable by a Büchi automaton [32].



(a) An example NFA on the alphabet $\Sigma = \{0, 1\}$



(b) DFA converted from NFA 4a

Figure 4: NFA-DFA translation *

* Converted using JFLAP tool [124]

Definition 2.4 (Regular expressions)
Regular expressions (Regex) are the simplest notations for describing a set of strings that are used in many applications such as pattern recognition, pattern matching, string searching as well as natural language processing. The simple version of its syntax is recursively defined as [96]

$$E = \text{true} \mid a \mid E_1 : E_2 \mid E_1 + E_2 \mid E^* \mid E^+. \tag{1}$$

If a regex term (E) *matches* a string, then the string is an element of that language $L(E)$ [96].

- The elementary regular expression is a single literal character in the alphabet $a \in \Sigma$ and the language of a is the set of the character, $L(a) = \{a\}$. The other symbols $(+, :, ^*, ^+)$ are the metacharacters.
- $L(\varepsilon) = \{\varepsilon\}$ whose only string is the empty string, or the length of its string is 0, and the latter denotes the empty set of strings.
- Two regular expressions can be concatenated to create a new regular expression: if E_1 matches string s and E_2 matches t then, $E_1 : E_2$ matches st, meaning that $L(E_1 : E_2) = L(E_1)L(E_2)$.
- Two regular expressions can be alternated to create a new regular expression: if E_1 matches string s and E_2 matches t, then $E_1 + E_2$ matches s or t, meaning that $L(E_1 + E_2) = L(E_1) \cup L(E_2)$.
- The meta-characters $^*, ^+$ represent repetition operators: E^* matches a sequence zero or more time (possibly different strings in each time), whereas E^+ matches one or more time.

  $L(E) = L(F)^*$, if $E = F$.
  $L(E) = L(F)$, if $E = F^*$.

Definition 2.5 (Regex Equivalence)
Let e, f, g *be regular expressions; if* $L(e) = L(f)$*, then* $e \equiv f$.

Equivalence laws based on fundamental rules of Kleene algebra, such as associativity, commutativity, idempotence, are listed as follows [97]. We use these simplifications for the stochastic regular expression presented in this thesis in the next chapters.

$$
\begin{aligned}
e + (f+g) &= (e+f) + g && \text{associativity of } + \\
e+f &= f+e && \text{commutativity of } + \\
e+e &= e && \text{idempotence of } + \\
e+0 &= e && 0 \text{ is an identity for } + \\
e:(f:g) &= (e:f):g && \text{associativity of } : \\
e:1 &= 1:e = e && 1 \text{ is an identity for } . \\
e:0 &= 0:e = 0 && 0 \text{ is annihilator for } : \\
e:(f+g) &= e:f + e:g && \text{distributivity} \\
(e+f):g &= e:g + f:g && \text{distributivity} \\
1 + e:e^* &= e^* \\
1 + e^*:e &= e^* \\
f + e:g \leqslant g &\Rightarrow e^*:f \leqslant g \\
f + g:e \leqslant g &\Rightarrow f:e^* \leqslant g
\end{aligned}
$$

In these relations, 1 and 0 are ε and $\emptyset$, respectively, and $\leqslant$ represents the *refined order*, meaning that:

$$e \leqslant f \Leftrightarrow e + f = f \text{ (refers to } \subseteq).$$

In the following, we list more useful expressions to simplify Regex terms [97] preserving the equivalence relationship:

$$
\begin{aligned}
e^* : e^* &= e^* \\
e^{**} &= e^* \\
(e^* : f)^* : e^* &= (e+f)^* && \text{denesting rule} \\
e : (f : e)^* &= (e : f)^* : e && \text{shifting rule} \\
e^* &= (e:e)^* + e:(e:e)^*
\end{aligned}
$$

2.1.1 *Linear Equations Solving: From FA to Regex*

A regular expression can be obtained from an FA using various techniques such as state elimination [40], equation solving [21]. In the following, we present the linear equation system representing a regex using Brzozowski's method [21]. The method transforms an automation to a regex by processing the following steps:

- Create a system of regular equations with one corresponding regular expression for each state in the given FA: Constructing the characteristic equations is therefore straightforward. For each state q_i in the FA, the equation for regex R_i is a

union of terms that are actually the union of outgoing action labels to the target states (corresponding R_i). More explicitly, the term aR_j for R_i represents a transition a from q_i to q_j. If R_i is a final state, ϵ is assigned as a term, which leads to a system of equations in the form:

$$\begin{aligned} R_1 &= a_1^1 R_1 + a_2^1 R_2 + \dots \\ R_2 &= a_1^2 R_1 + a_2^2 R_2 + \dots \\ R_3 &= a_1^3 R1 + a_2^3 R_2 + \dots + \epsilon \\ \dots \\ R_n &= a_1^n R_1 + a_2^n R_2 + \dots + \epsilon \end{aligned} \tag{2}$$

- Solve the system using *Arden's lemma* [4]:

  Given an equation of the form $X = AX + B$ where $\epsilon \notin A$,
  the equation has the solution $X = A^*B$.

- The regex equivalent to the initial state will be the regex representing the given FA.

Example 2.3 (From FA to regex)
Let us consider the FA demonstrated in Figure 3. As the precondition of this approach, we first separate the initial and final states. The new version is presented in Figure 5.

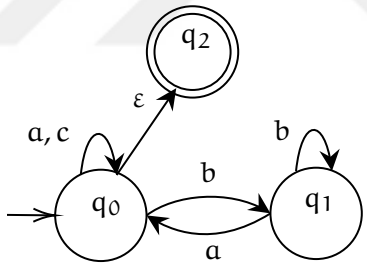


Figure 5: Changed FA from Figure 3

Hence, the equation system is constructed as follows: All the states q_0, q_0, q_1 and q_2 are related to the outgoing states with corresponding transition label. The final state is additionally connected with ε.

$$\begin{aligned} q_0 &= a : q_0 + c : q_0 + b : q_1 + \varepsilon : q_2 \\ q_1 &= a : q_0 + b : q_1 \\ q_2 &= \varepsilon \end{aligned} \tag{3}$$

q_2 is eliminated because of the ε transition:

$$\begin{aligned} q_0 &= a : q_0 + c : q_0 + b : q_1 \\ q_1 &= a : q_0 + b : q_1 \end{aligned} \tag{4}$$

By distributivity in q_0:

$$\begin{aligned} q_0 &= (a+c) : q_0 + b : q_1 \\ q_1 &= a : q_0 + b : q_1 \end{aligned} \tag{5}$$

By commutativity of +

$$\begin{aligned} q_0 &= (a+c) : q_0 + b : q_1 \\ q_1 &= b : q_1 + a : q_0 \end{aligned} \tag{6}$$

By Arden's Rule, q_1 *is eliminated from recursion.*

$$\begin{aligned} q_0 &= (a+c) : q_0 + b : q_1 \\ q_1 &= b^* : a : q_0 \end{aligned} \tag{7}$$

q_1 *is eliminated by substitution.*

$$q_0 = (a+c) : q_0 + b : b^* : a : q_0 \tag{8}$$

By distributivity in q_0*:*

$$q_0 = ((a+c) + b : b^* : a) : q_0 \tag{9}$$

By Arden's rule, q_0 *is eliminated from recursion and finally,* q_0 *represents the regex from the automation.*

$$q_0 = ((a+c) + b : b^* : a)^* \tag{10}$$

By associativity, we obtain the language of regular expression provided in Figure 3 as follows:

$$q_0 = (a + c + b : b^* : a)^* \tag{11}$$

2.1.2 *Parsing Regex*

The process of a given syntax to analyze the grammar is simply called *parsing*. There exist two broad approaches called *top-down* in which the abstract syntax tree (AST) is built from the root downwards to the leaves and *bottom-up* in which the AST is built from the leaves upward to the root node [4]. The leaf nodes and the intermediate nodes including the root node represent the terminal and non-terminal symbols, respectively.

In Figure 6, we present an AST for a given regex $(a : b)^* : a : b : a + b : b : a$ that is built in a bottom-up fashion using the syntax (formal grammar in (1)).

The order of the numbers in the AST in Figure 6 demonstrates a LL (left-most parser) terminal processing, which is a commonly used type of bottom-up parsing. Hence, the numbered leaves are the terminals of AST, and the non-terminal nodes E are constructed using the grammar rules (1) such that the operator precedence order is $^* >: > +$.

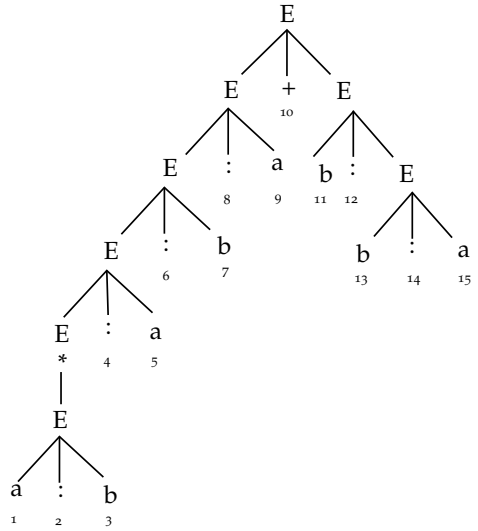


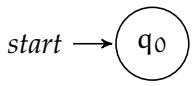
Figure 6: Abstract syntax tree for the regex $(a : b)^* : a : b : a + b : b : a$

2.1.3 *Constructing FA from Regex*

There are various construction techniques that produce FA from regular expressions. Popular examples of these techniques are Thompson's and Glushkov's constructions. For instance, *Thomson construction* algorithm inductively constructs a NFA [115, 138].

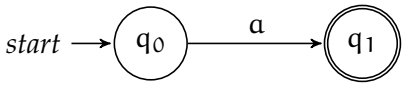
Definition 2.6 (Thompson's NFA)
A NFA is built inductively for every operator in Regex [138].

- An empty set obviously constructs no NFA. $E = \emptyset$.

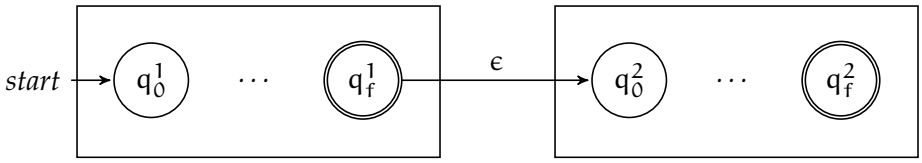


- An epsilon transition is terminated immediately. $E = \epsilon$.

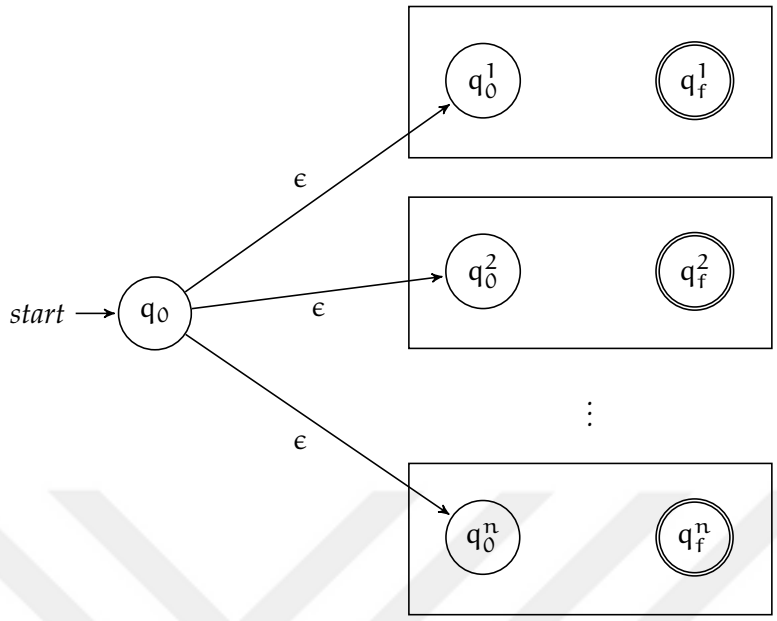
- The NFA is constructed for a single character $E = a$ as follows:



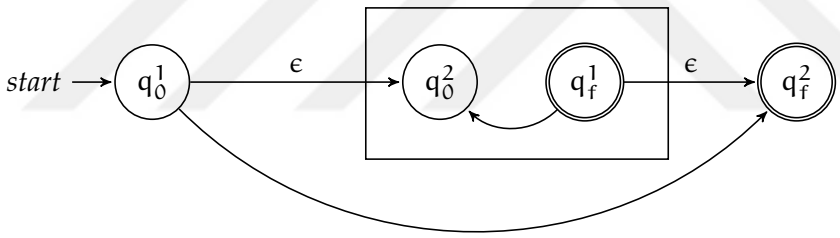
The NFA for the concatenation $E = E_1 : E_2$ connects the final transition of the E_1 machine to the start state of the E_2 machine.



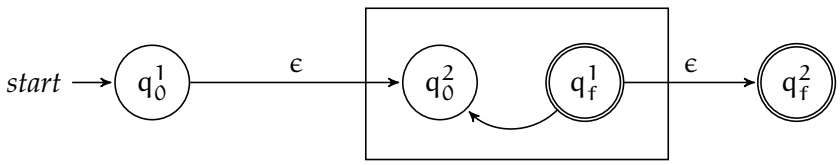
A new start state is added to each machine $E_1, E_2, \cdots E_n$ for the NFA of the alternation $E = E_1 + E_2 + \cdots E_n$.



The NFA for E^* uses the same alternation but adds a loop of E machine back to the start state of E including ϵ.



Finally, the NFA for E^+ is similar to E^* that adds an alternation but with a loop of E machine back to the start state of *itself* excluding ϵ execution.



The Thompson's construction [138] was designed in 1968 for the purpose of text searching for the given regular expressions. The automata are inductively constructed and have the following properties:

- There is only one start state, that is there are no transitions entering it.
- There is one final state, that is only entering.
- Each state has at most two transitions, leaving it and at most two transition entering it.

- The size of the machine is at most **three times of the size of the given regular expression**.

Hence, we also provide the summary of Glushkov's automation and its construction method that eliminates some limitations of the NFA.

Definition 2.7 (Glushkov Automata)
The ϵ-free NFA that is constructed by the following algorithm is a Glushkov automata [65].

The original construction of the automata is based on the *first, last, and follow* sets of the positions in the given regular expression. The appearances, from left to right, of the Σ-symbols in a regular expression are numbered from one to the total number of appearances. The appearances are called positions, and we modify a regular expression E to obtain a new regular expression $\overline{E}$ in which each symbol is replaced by its position; thus, if there are n appearances, $\overline{E}$ is a regular expression over $\{1, \ldots, n\}$. If $a \in \Sigma$ is at a position i, then we say that i corresponds to a.

Let us explain the definition through an example. Let $E = (AT|GA)((AG|AAA)^*)$ be a regular expression defined on the alphabet $\Sigma = \{A, G, T\}$.

The positions are defined by linearization step provided in the following paragraph. Linearization: The regular expression is marked with positions for the symbols and denoted as $\overline{E} = (A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)$, and the position set is $\{1, \ldots, 9\}$. The three sets of positions are defined as follows:

- $\mathtt{first}(E)$ is the set of all positions that can begin in a string in $L(\overline{E})$. Then, $\mathtt{first}(\overline{E}) = \{1, 3\}$.
- $\mathtt{last}(E)$ is the set of all positions that can end in a string $L(\overline{E})$. $\mathtt{last}(\overline{E}) = \{2, 4, 6, 9\}$.
- $\mathtt{follow}(i, E)$ is the set of all positions in $L(\overline{E})$ than an reachable from position i. e.g. $\mathtt{follow}(\overline{E}, 6) = \{7, 5\}$.

The constructed automation based on this algorithm is demonstrated in Figure 7. The Glushkov's automation has the following properties:

- There is only one exiting start state.
- **It has no ε transition**
- For each state p, all transitions into p have the same label.
- The size of the Glushkov machine of a regular expression E is, in the worst case, $\mathcal{O}(|E|^2)$.

2.1.4 *Pattern Matching of Regex Using NFA*

We begin with a simple definition of pattern matching in the scope of the thesis.

Definition 2.8 (Pattern Matching)
For a given word (string) w of length n and a regex E (considered pattern), pattern matching determines whether $w \in L(E)$. In a FA, this can be interpreted whether a (finite) path is a run of FA [35].

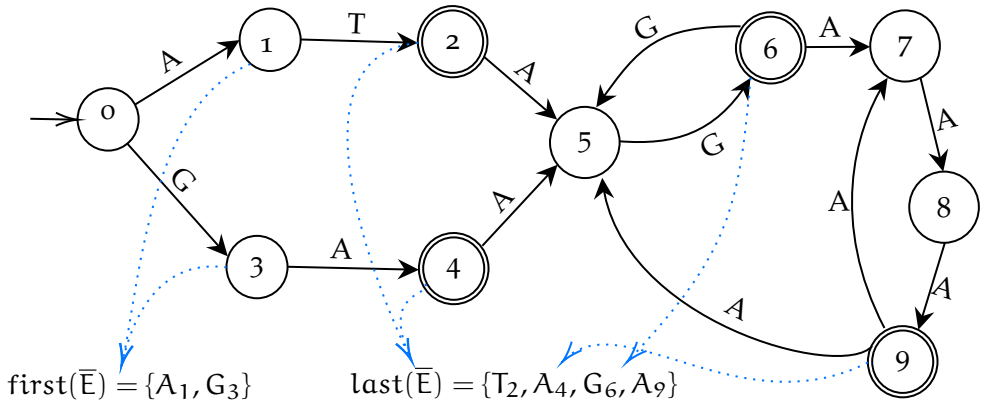


Figure 7: FSM constructed from the regex $E = (AT|GA)((AG|AAA)^*)$

One of the approaches finding the matches on Regex is building a NFA which can be directly constructed from the syntax tree (provided in subsection 2.1.3). Such approach traverses the paths of NFA until the final states.

Example 2.4 (String matching)

Let $E = b : a : b : b : a + b : a : b : b : b + a$ *be a regular expression defined on the alphabet* $\Sigma = \{a, b\}$.

The search string for matching is given as babbbb. The constructed FA from the regex E is provided in Figure 8 by presenting the steps of the matching algorithm. In a single processing of a path, the algorithm follows through the path until reaching the final state. In case of failing the matching, it backtracks (e.g., step 5 in Figure 8) and starts a new path. Obviously, a more efficient way of searching is to run FAs in parallel to all outgoing transitions from the current state.

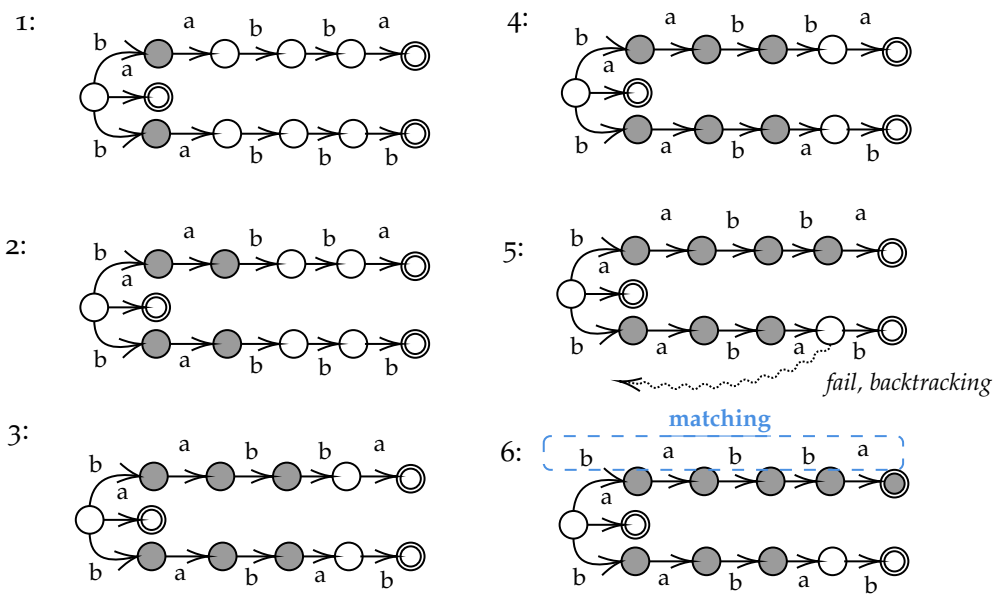


Figure 8: Example of pattern matching

We demonstrate the steps of the parallel approach rather than a single path traversing by coloring the states until reaching the final state. In this case, the matching string is obtained in the upper path as shown in Figure 8 [35].

2.1.5 *Model Checking Automata of Linear Properties*

Finite automata can be used to model concurrent and interactive systems. Either the state Q or the alphabet Σ can then represent the states of the modeled system. In model checking, both the system model and the specification are represented in the same way using a Kripke structure [32].

A Kripke structure directly corresponds to an ω-regular automaton, where all the states are accepting. Then, the set of behaviors of a system M is the language $\mathcal{L}(\mathcal{A})$ of the corresponding automaton $\mathcal{A}$.

Definition 2.9 (Kripke structure)
Specifically, a Kripke structure is a tuple (S, R, S_0, L) *where* S *is the set of states,* R *is the relation set, and* $S_0 \subset S$ *is the set of initial states.* $L : S \to 2^{AP}$ *can be transformed into an automaton* $\mathcal{A} = \{\Sigma, S \cup \{\iota\}, \delta, \{\iota\}, S \cup \{\iota\}\}$, *where* $\Sigma = 2^{AP}$, *and AP represents the atomic propositions. In this case,* $(s, \alpha, s') \in \delta$ *for* $s, s' \in S$ *if and only if* $(s, s') \in R$ *and* $\alpha = L(s')$. *In addition,* $(\iota, \alpha, s) \in \delta$ *if and only if* $s \in S_0$ *and* $\alpha = L(s)$ *[32].*

The specification formula is also defined as an automaton $\mathcal{S}$, over the same alphabet. Then, $\mathcal{L}(\mathcal{S})$ is the set of allowed behaviors. Each edge may represent several transitions, where each transition corresponds to a truth assignment for AP that satisfies the boolean expression.

For example, let $AP = \{a, b, c\}$ be a set of propositions, an edge labeled $\neg a \wedge c$ matches the transitions labelled with $\{b, c\}$ and $\{c\}$. Since the goal is to find the sets of propositions that include c and do not include a, but may or may not include b [32].

According to the underlying idea of the linear model checking, the system $\mathcal{A}$ satisfies the specification $\mathcal{S}$ when

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{S}). \tag{12}$$

It means that each behavior of the modelled system is among the behaviors that are allowed by the specification. Let $\overline{\mathcal{L}(\mathcal{S})}$ be the language of $\mathcal{L}(\mathcal{S})$'s complement: $\Sigma^\omega - \mathcal{L}(\mathcal{S})$. Then (12) is rewritten as

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset. \tag{13}$$

Hence, there is no behavior of $\mathcal{A}$ that is disallowed by $\mathcal{S}$. Therefore, if the intersection is not empty, there exists a counterexample that corresponds to the behavior of the intersection. In other words, there exists an automaton that accepts exactly the intersection automation and an automaton that recognizes the complement of the language of the given automaton. The details of computing the complement of a Büchi automaton and how to construct an automaton that recognizes the intersection of two languages accepted by a pair of Büchi automata can be found in [31]. The formulation of the correctness criterion in (13) suggests the following model checking procedure [32]:

- Complement the automaton $\mathcal{S}$, that is, construct an automaton $\mathcal{S}$ that recognizes the language $\overline{\mathcal{L}(\mathcal{S})}$.

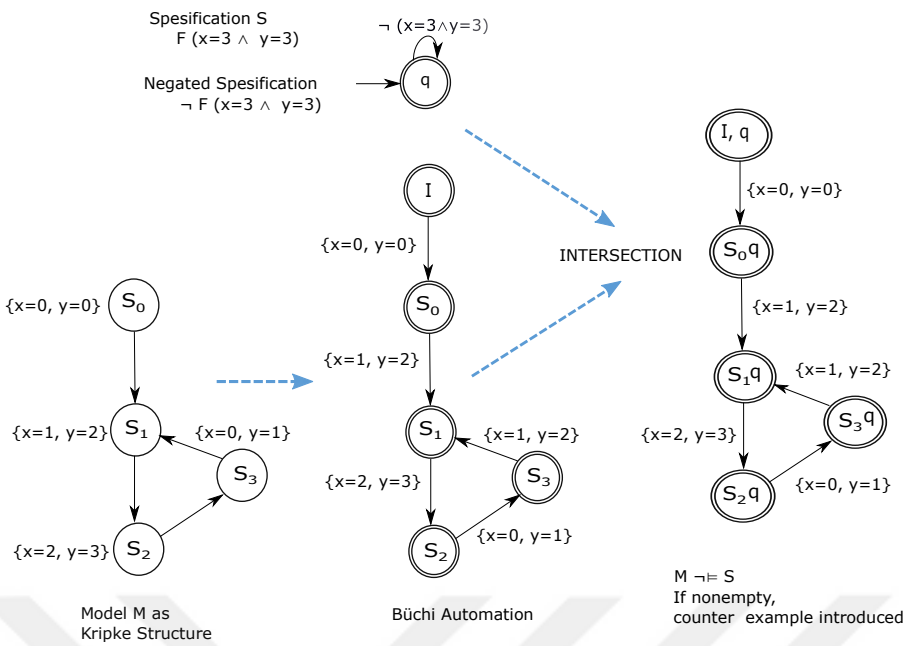


Figure 9: Automata-based LTL model checking in a nutshell with an example [22]

- Construct the automaton that accepts the intersection of the languages $\mathcal{L}(\mathcal{A})$ and $\overline{\mathcal{L}(\mathcal{S})}$.

If the intersection is empty, the specification $\mathcal{S}$ holds for $\mathcal{A}$. Otherwise, a counterexample has to be provided. An overview of the procedure is provided in Figure 9.

2.1.6 *Action-based Computation Tree Logic (ACTL*)*

ACTL* is introduced in the study where the model checking for state-labeled and transition-labeled systems is compared [37]. The syntax of ACTL* is described on action-labeled transition systems recursively as follows:

$$\varphi := \mathrm{True} \mid \neg\varphi \mid \varphi \wedge \varphi' \mid \exists\varphi \mid \varphi\, \mathcal{U}\, \varphi' \mid \mathcal{X}_a \varphi \mid \mathcal{X}\varphi \tag{14}$$

where φ is a formula executed on the runs of the system and a is an action.

We briefly recall the definitions required for the ACTL* semantics.

Definition 2.10 (Labeled transition systems)
A labelled transition system is a tuple $(S, \mathrm{Act}, \rightarrow)$ *[37] where*

- S *is a finite set of states,*
- $\mathrm{Act} = \{\alpha_0, \ldots, \alpha_n\}$ *is a finite, non-empty set of actions,*
- $\rightarrow \subseteq S \times (\mathrm{Act} \cup \epsilon) \times$ *is the transition relation, and any element of* $\rightarrow$ *is called a transition* (s, α, s').

Definition 2.11 (A run of the system)
A sequence $(q_0, \alpha_0, q_1)(q_1, \alpha_1, q_2)\ldots$ *is called a* ***path*** *from* q_0. *If a path* π *is infinite or ends in a final state, then it is called a fullpath [37]. A* ***run*** $\rho = (q, \pi)$ *is a pair from* $q \in Q$, *where* π *is a path from* q, *the first state of* ρ *is* q *i.e.,* $\mathrm{first}(\rho) = s$, *and* $\mathrm{path}(\rho) = \pi$. *If a* ***run*** θ *is a proper suffix or a suffix for* ***run*** ρ, *then the suffix is denoted as* $\rho < \theta$ *and* $\rho \leqslant \theta$ *respectively.*

The semantics is given by satisfaction relations between a run ρ and a path formula φ based on the definitions above [37].

$\rho \models \neg\varphi$ iff $\rho \not\models \varphi$
$\rho \models \varphi \wedge \varphi'$ iff $\rho \models \varphi$ and $\rho \models \varphi'$
$\rho \models \exists\varphi$ iff there exists a run $\theta \in \mathtt{run}(\mathtt{first}(\rho))$ such that $\theta \models \varphi$
$\rho \models \varphi\, \mathcal{U}\, \varphi'$ iff there exists a θ with $\rho \leqslant \theta$ such that $\theta \models \varphi\prime$ and for all $\rho \leqslant \eta \leqslant \theta : \eta \models \varphi$
$\rho \models \mathcal{X}\varphi$ iff there exists q, arbitrary action α, $q\prime$, θ such that $\rho = (q, (q, \alpha, q\prime))\theta$ and $\theta \models \varphi$
$\rho \models \mathcal{X}_a\varphi$ iff there exists s, a, $s\prime$, θ such that $\rho = (s, (s, a, s\prime))\theta$ and $\theta \models \varphi$ (15)

2.2 NOTES ON PROBABILITY THEORY AND QUANTITATIVE VERIFICATION

This section briefly provides the terminology and concepts of probability theory as well as probabilistic model checking; nevertheless, it is not a complete introduction to probability theory. We refer the interested readers to [45] and [91] for the details.

Definition 2.12 (Measurable set)
A measurable space is a collection ω, satisfying the following conditions:

- $\emptyset \in \omega$,
- *For any* $A, B \in \omega$, $A/B \in \omega$,
- *For any* $A_1, A_2, \ldots \in \omega$, $\cup A_i \in \omega$,

The elements of ω are called measurable sets [45].

Measures are a generalization of the length of volume concepts of Euclidean geometry into other spaces and form the basis of probability theory. In this part, we briefly explain what a space means for a measurable set and the construction of *measurable* spaces.

A σ algebra Ω on a set S is a collection of subsets of S containing $\emptyset$ and closed under complement, countable union, and intersection. A pair (S, Ω) is called a *measurable space*. For a measurable space (S, Ω), we say that a subset $A \subseteq S$ is *measurable* if it is in B. For applications in probability theory, the elements of S and Ω are often called *outcomes* and *events*, respectively [45].

Definition 2.13 (Probability Function)
For a given sample space Ω, F is a ***probability function*** $P : \mathcal{F} \to [0, 1]$ *satisfies the following properties, where F is σ-algebra:*

- $P(A) \geqslant 0$ *for* $A \in F$,
- $P(\Omega) = 1$, *and*
- $P(\cup_{i=1}^{\infty} A_i) = \Sigma_{i=1}^{\infty} P(A_i)$ *for pairwise disjoint* $A_i \in F$ *[98].*

Definition 2.14 (Probability Space)
A probability space is a triple (Ω, F, P), where Ω is an arbitrary set called sample space and ξ is a σ-algebra on Ω of which members are ***measurable set*** *and* $P : \mathcal{F} \to [0, 1]$ *is a* ***probability function****.*

Definition 2.15 (Randomness)
*A random variable x is a **measurable** function $x : \Omega \to I$ to some I. Elements of I are random elements, usually selected from $\mathbb{R}$ [98].*

Example 2.5 (Probability space, probability function)
Let us consider a fair dice tossing two times. The outcome is all possible six facets of the dice; hence, it is (sample space) $\Omega = \{1, 2, 3, 4, 5, 6\}$. The outcome of two dice is every combination of six facets, that is $\Omega = \{(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,2), (2,3), \cdots, (6,6)\}$.

The probability function in this example is the tossing one of the combinations from $(1,1)$ to $(6,6)$; thus, $P(\{\}) = 0$, $P(\{1,1\}) = P(\{1,2\}) \cdots P(\{6,6\}) = \frac{1}{36}$, and $P(\{(1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,2), (2,3), \cdots, (6,6)\}) = 1$.

Definition 2.16 (Measures)
A measure on (S, Ω) is a countable additive map $\mu : \Omega \to \mathbb{R}$. Being a countable additive map stipulates that if $A_i \in \Omega$ is a countable set of pairwise disjoint events, then $\mu(\cup_i A_i) = \Sigma_i \mu(A_i)$. Equivalently, if A_i is a countable chain of events, that is $A_i \subseteq A_j$ for $i \leqslant j$ then $\lim_i \mu(A_i)$ exists and is equal to $\mu(\cup_i A_i)$. A measure is a probability measure if $\mu(A) \geqslant 0$ for all $A \in \Omega$ and $\mu(S) = 1$. By convention, $\mu(\emptyset) = 0$.

For every $a \in S$, the *Dirac* measure on a is the probability measure:

$$\begin{cases} \delta_a(A) = 1 & a \in A \\ 0 & a \notin A \end{cases}$$

A measure is *discrete* if it is a countable weighted sum of Dirac measures [98].

Definition 2.17 (Stochastic process)
*A stochastic process is a family of random variables $\{X_t | t \in T\}$ for a probability space $(\Omega, \mathcal{F}, \mathcal{P})$ for each $X_t : \Omega \to S$, and $S = \{s_1, s_2, \cdots\}$ is a finite or countable set called **state space**. A stochastic process is **discrete-time** if $T = \mathbb{N}$ or **continuous-time** if $T = \mathbb{R}_{\geqslant 0}$ [98].*

Example 2.6 (Weather forecast)
Let $S = \{\mathtt{sun}, \mathtt{rain}\}$, and the time is modeled as discrete meaning that a random variable is a set for each day [98].

- *X_0 is the weather today,*
- *X_i is the weather on ith days,*
- *The sample space $\Omega = S^\infty$ so that each X_i maps a sample $\omega = \omega_0 \omega_1 \cdots$ on the respective state at time i, $(X_n)(w) = w_n \in S$.*

Definition 2.18 (Probabilistic language (p-language))
Any finite, non-empty set Σ is called alphabet. Σ^* denotes all words over Σ, and Σ^+ denotes all words over Σ excluding the empty word ε.

We revise the definition of the probabilistic language (p-language) described in [54]. A p-language is a probabilistic value map that assigns probability measure to each trace s in Σ^*, where Σ is the alphabet of the language. This measure determines the probability of that trace's occurrence that is followed by the termination event σ_Δ.

Formally, a p-language L is a probability measure on a measurable space $(\Omega, \mathcal{F})$ such that it is a unit interval valued map $L : F \to [0, 1]$, where $\mathcal{F}$ is the σ—algebra generated by $\{< s >$ such that $s \in \Omega\}$ and $\Omega = \Sigma_\Delta^{*\sigma} \cup \Sigma$. A p-language is a σ—algebra; therefore, it satisfies the following properties (σ—additivity).

- $L(\varepsilon) = 1$
  (A zero length trace is always possible in a system, its probability measure must be one).

- $\Delta(L) = L - \sum_{\sigma \in \Sigma} L(s\sigma) > 0$
  (The cumulative probability of all traces sharing a common prefix should not exceed the probability of the prefix itself).

where $L(s\sigma_\Delta)$ *represents the probability of termination following* s*,* $L(s)$ *represents the probability of occurrence of* s*, and* $\sum_{\sigma \in \Sigma} L(s\sigma)$ *represents the probability of continuous operation beyond* s*.*

Given a pair of p-languages $L_1, L_2 \in \mathcal{L}$, and $p \in [0, 1]$, three operators are defined on the p-language:

1. **Choice**: $L_1 +_p L_2 := pL_1 + p'L_2$, where $p' := 1 - p$.

2. **Concatenation**: $\forall s \in \Sigma : L_1 :_p L_2 = L_1(s) + p \sum_{t<s} \Delta(L_1)(t) L_2(t^{-1}s)$.
   Stating differently, the composite system executes a trace s of L_1 in the first run, or it either executes the entire trace s in that way, or terminates after executing a prefix t of s. Then it executes the remainder of the trace $t^{-1}s$ of L_2 with probability p.

3. **Concatenation closure:** It is the infimum fixed point of concatenation function $L_{:p}L$. We refer to [54] for further theorems and proofs of the operations and partial order property.

Such systems whose state evolution depends entirely on the occurrence of asynchronous discrete events over time have event-driven discrete states. The set of all p-languages forms an infinite semi-lattice and a complete partial order with respect to the natural ordering of elements. Thus, recursive functions can be defined via fix-points on this set. Various operators are defined between probabilistic languages (automata) which can be used to build complex systems from simpler systems. In particular, regular operators (union, concatenation, and concatenation closure) preserve finiteness, ordering, and least upper bounds of chains [99].

The notion of regularity is defined closed under the operations of regular language operators. According to the mentioned definition, Σ denotes the universe of events, and the set Σ^* is the set of all finite length event sequences, which are called traces, including the zero length trace denoted by an empty string ϵ. Any subset of Σ^* is a language such that, given traces s and t, if s is a *prefix* of t. If the language is *prefix closure*, then the set of all prefixes is equal to that language.

Example 2.7 (Bernoulli process in p-language)
A trivial example is presented on language $\mathcal{L}$*. Let* L *be a p-language describing the Bernoulli process where each experiment has two outcomes* a *and* b *with probabilities with* p *and* $1 - p$*, respectively.* L *is defined on the alphabet* $\Sigma = \{a, b\}$ *and denoted as* $L(s) = p^{\#(a,s)}.(1-p)^{\#(b,s)}$ *where* $\#(a, s)$ *represents the number of occurrences in the trace* s *[54].*

A Stochastic regular expression (SRE) represents a very similar algebra to p-language and includes the same operations as described in Chapter 4. Additionally, it enables the system representation not only by means of measures but also syntactically using regular expressions.

Definition 2.19 (Probabilistic Rabin automata (PRA))
A PRA consists of five components

- *A set of states* Q,
- *A non-empty set* $Q^0 \in Q$ *of start states,*
- *An action set* Act,
- *A transition relation* $\rightarrow \in Q \times \mathrm{Act} \times P$.
- *A set of accepting states* $F \subseteq Q$

A PRA is a labeled transition system with finite states and probabilistic transitions among them. Any string s is accepted with a certain probability [125].

Let $\mathcal{A} = (\Sigma, Q, q_0, P, F)$ be a PRA where Σ is the alphabet, Q is the state space. $q_0 \in Q$ represents the initial state, and $F \subseteq Q$ is a set of final states. $P : Q \times Q \times \Sigma \cup \{\epsilon\} \rightarrow [0, 1] \cap \mathbb{Q}$ is the probability function that assigns a probability to each transition. The probability function may be partial; if $P(q, q', a)$ is undefined, it means that there is no transition from q to q' with the character a. This is semantically equivalent to $P(q, q', a) = 0$; therefore, we can always consider P as a total function without loss of generality [125].

Example 2.8 (Example of a PRA)
An example PRA defined on the alphabet $\Sigma = \{a, b, c, d\}$ *is demonstrated in Figure 10. The initial state set is* $Q^0 = \{q_0\}$*, the final state set is* $F = \{q_3\}$*, and the transition relation is* $\rightarrow = \{(q_0, a, 0.01), (q_0, b, 0.99), (q_1, c, 0.8), (q_1, c, 0.15), (q_1, d, 0.05)\}$*. The summation of all distributions of the actions that are outgoing from the same state will be* 1.0*, e.g.,* $0.01 + 0.99 = 1$ *for all outgoing actions of the state* q_0*.*

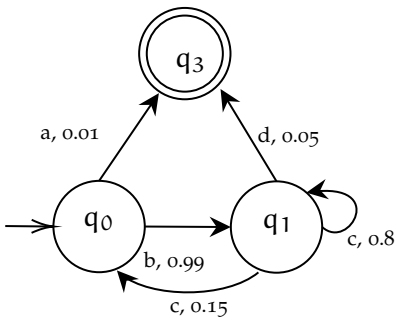


Figure 10: An example PRA

2.3 CONVENTIONAL PROBABILISTIC MODEL CHECKING

There exist various methods for the probabilistic (a.k.a. quantitative) verification of systems including statistical model checking, simulations, probabilistic model checking [91]. Probabilistic model checking is a mathematical reasoning technique that automatically performs probabilistic assessments, such as non-functional requirement analysis, and has been used successfully in recent years [103].

In addition to the traditional model checking [32], probabilistic model checking allows the analysis on the randomized, sequential and distributed computing. It is

broadly applied in reliability and performance engineering as well as dependability analysis often using Markov chains and Markov decision processes (MDP) as a model.

Discrete-time Markov chains are essentially a state-transition system augmented with probabilities in which time progresses in discrete intervals. The next state at each point in time is specified by a discrete probabilistic distribution from source to target states.

Definition 2.20 (Discrete-time Markov chain (DTMC))
A DTMC over a set of atomic propositions AP is a tuple $\mathcal{M} = (\mathcal{S}, \boldsymbol{P}, \iota_{init}, AP, L)$ *where*

- $\mathcal{S}$ *is a countable, non-empty set of states,*
- $\boldsymbol{P} : \mathcal{S} \times \mathcal{S} \to [0, 1]$ *is the transition probability function such that for all states s:* $\sum_s' \in \mathcal{S}, \boldsymbol{P}(s, s') = 1$,
- $\iota_{init} : \mathcal{S} \to [0, 1]$ *is the initial distribution, such that* $\sum_{s \in \mathcal{S}} \iota_{init}(s) = 1$, *and*
- *AP is a set of atomic propositions, and* $L : \mathcal{S} \to 2^{AP}$ *a labelling function [91].*

$\mathcal{M}$ is called finite if $\mathcal{S}$ and AP are finite. The size of $\mathcal{M}$ is the number of states plus the number of pairs $(s, s') \in \mathcal{S} \times \mathcal{S}$ with $\mathbf{P}(s, s') > 0$. The transition probability function $\mathbf{P}$ specifies for each state s the probability $\mathbf{P}(s, s')$ of moving from s to s' in one step by a single transition where $\mathbf{P}$ has to be a distribution [9].

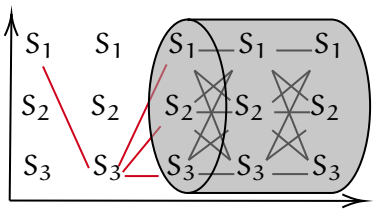
The atomic propositions and the labelling function $\mathcal{L}$ are the same as for transition systems. In a Markov chain, states are vertices, and there is an edge from s to s' if and only if $\mathbf{P}(s, s' > 0)$.

Definition 2.21 (σ-Algebra of a Markov Chain)
The paths in Markov chains $\mathtt{Paths}(\mathcal{M})$ *are maximal (infinite) paths in the underlying digraph; formally,* $\pi = s_0 s_1 s_2 \cdots \in \mathcal{S}^\omega$, *such that* $\boldsymbol{P}(s_i, s_i + 1) > 0$ *for all* $i \geqslant 0$. *The σ-algebra of a Markov chain is generated by the cylinder sets spanned by the finite path fragments in* $\mathcal{M}$ *[98].*

Definition 2.22 (Cylinder Set)
For $s_0 \cdots s_n$, *the cylinder* $\mathtt{Cyl}(s_0 \cdots s_n) = s_0 \cdots s_n \omega \in \Omega$ *is called a* ***cylinder set*** *[91]. The cylinder set is spanned by the finite path* $\hat{\pi}$ *thus, it consists of all infinite paths that start with* $\hat{\pi}$. *The cylinder set of* $\hat{\pi} = s_0 \cdots s_n \in \mathtt{Paths}_{fin}(\mathcal{M})$ *is defined as*

$$\mathtt{Cyl}(\hat{\pi}) = \{\pi \in \mathtt{Paths}(\mathcal{M}) \mid \hat{\pi} \text{ is a prefix of } \pi\} .$$

Example 2.9 (Visual demonstration of a cylinder set)
Let $S = \{s_1, s_2, s_3\}$ *be then the cylinder set from* s_1 *to* s_3 *is demonstrated below because* $s_1 s_3$ *is a prefix for all the paths in the demonstrated cylinder [98].*



2.3.1 *Reachability Probabilities*

The reachability analysis (e.g., $\Diamond s$) is one of the common reasoning techniques to compute the probability of reaching a certain set of states. This set may represent a set of certain bad states which should be visited only with some small probability, or dually, a set of good states which should rather be visited frequently [91].

Formally, the reachability probability is calculated by the following probability measure reaching a goal G:

$$\Pr(s \models \Diamond G) = \Pr_s(\Diamond G) = \Pr\{\pi \in \mathrm{Paths}(M_s) \mid \pi \in \Diamond G\}$$

In this case, the set Pr is the unique probability distribution on all sets of infinite paths that are countable (disjoint) unions and/or complements of cylinder sets.

$$\Pr(\mathrm{Cyl}(s_0 \cdots s_n)) = \Pi_{0 \leqslant i < n} P(s_i, s_{i+1}) \text{ where } n > 0 \text{ and } P(s_0) = 1 \text{ iff } s_0 = s_1$$

The reachability probability on a state to reach a goal is characterized by the following linear equation system. Let variable $x_s = \Pr(s \models \Diamond G)$ for any state s and $\mathrm{Pre}(G)$ is the set of states in Σ from which G is reachable, then

$$\begin{aligned} x_s &= 0 && \text{if } s \notin \mathrm{Pre}(G) \\ x_s &= 1 && \text{if } s \in G \\ x_s &= \Sigma_{\alpha \in \Sigma \setminus G}\ P(s,\alpha) \cdot x_\alpha + P(s,\beta) && \text{otherwise} \end{aligned} \tag{16}$$

Example 2.10 (Computing Reachability Probability)
We demonstrate an example Markov chain of a simple communication protocol in Figure 11. The reachability probability of the action "delivered" is computed as follows [9]. An example of a path $\pi = (\mathrm{start\ try\ lost\ try\ lost\ try\ delivered})$. *The path fragments to the label "delivered" has the pattern*

$$\pi_n = \mathrm{starttry}(\mathrm{losttry})^n \mathrm{delivered}, \textit{ with } n \in \mathbb{N}.$$

Thus,

$$\Pr(\Diamond \mathrm{delivered}) = \Sigma_{n=0}^{\infty} \left(\frac{1}{10}\right)^n \cdot \frac{9}{10} = \frac{\frac{9}{10}}{1 - \frac{1}{10}} = \frac{\frac{9}{10}}{\frac{9}{10}} = 1.$$

This technique is considered to be less efficient. On the other hand, solving the linear equation without considering the infinite sum is a widely used technique as presented below for the same example [9] (i.e., using the formula (16)).

$$\begin{aligned} x_{\mathrm{start}} &= x_{\mathrm{try}} \\ x_{\mathrm{try}} &= \frac{1}{10} \cdot x_{\mathrm{lost}} + \frac{9}{10} \cdot \overbrace{x_{\mathrm{delivered}}}^{=1} \\ x_{\mathrm{lost}} &= x_{\mathrm{try}} \end{aligned}$$

The equations can be represented in the probability matrix as

$$\begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & -\frac{1}{10} \\ 0 & -1 & 1 \end{pmatrix} \cdot x = \begin{pmatrix} 0 \\ \frac{9}{10} \\ 0 \end{pmatrix}$$

using

$$x = A \cdot x + b \implies (I - A) \cdot x = b \text{ with } I \text{ is the identity matrix.}$$

The unique solution is $x_{start} = x_{try} = x_{lost} = 1$. *Hence, reaching the state "delivered" is almost sure from any state.*

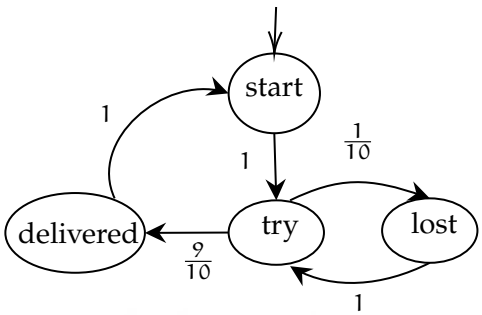


Figure 11: Markov chain representing a communication protocol [9]

2.3.2 *Probabilistic Computation Tree Logic*

Probabilistic Computation Tree Logic (PCTL)[79] is a branching-time temporal logic for describing properties of DTMCs. To this end, PCTL extends the non-probabilistic Computation Tree Logic [9] with a probabilistic operator P.

Definition 2.23 (Syntax of PCTL)

$$\begin{aligned} \Phi &:= \mathtt{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathcal{P}_{\bowtie p}(\varphi) \\ \varphi &:= \mathcal{X}\Phi \;\Big|\; \Phi_1 \,\mathcal{U}\, \Phi_2 \;\Big|\; \Phi_1 \,\mathcal{U}^{\leqslant k}\, \Phi_2 \end{aligned} \tag{17}$$

where:

- $a \in AP$ *is an atomic proposition,*
- $\bowtie \; \in <, \leqslant, \geqslant, >$,
- $k \in \mathbb{N} \cup \{\infty\}$, *and*
- $p \in [0, 1]$ *is a probability bound.*

As a consequence, probabilistic properties can be expressed in the logic PCTL, which extends the temporal logic CTL with the ability to reason quantitatively.

The PCTL model checking algorithm takes a probabilistic model (e.g., DTMC) and a PCTL formula and produces the set of the states satisfying the formula, $Sat(\Phi) = \{s \in S \mid s \models \Phi\}$.

Example 2.11 (Example of a PCTL model checking on Next Formula)
Let us consider the example DTMC provided in Figure 11. Hence, the following process is executed for a PCTL formula [9]

$$\mathtt{Prob}_{\geqslant 0.99}(\mathcal{X}(\neg \mathtt{try} \vee \mathtt{delivered})).$$

- *Satisfactory states are identified.*

$$\begin{aligned}\mathrm{Sat}(\neg \mathrm{try} \vee \mathrm{delivered}) &= (S \setminus \mathrm{Sat}(\neg \mathrm{try}) \cup \mathrm{Sat}(\mathrm{delivered})) \\ &= (\{\mathrm{start}, \mathrm{delivered}, \mathrm{try}, \mathrm{lost}\} \setminus \{\mathrm{try}\} \cup \{\mathrm{delivered}\}) \\ &= \{\mathrm{start}, \mathrm{delivered}, \mathrm{lost}\}\end{aligned}$$

- *The probability matrix is constructed and multiplied by the values of the satisfactory states.*

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{10} & \frac{9}{10} \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

- *Hence, the result of the formula*

$$\begin{aligned}&\mathrm{Prob}_{(\mathcal{X}(\neg \mathrm{try} \vee \mathrm{delivered}))} = [1, 0, 1, 1] \\ &\mathrm{Sat}(\mathrm{Prob}_{\geqslant 0.99}(\mathcal{X}(\neg \mathrm{try} \vee \mathrm{delivered}))) = \{\mathrm{try}, \mathrm{delivered}, \mathrm{lost}\}\end{aligned}$$

The more sophisticated formulas, such as unbounded and bounded $\mathcal{U}$ formula, requires pre-computation algorithms to find reachable states. The identification of the strongly connected components (SCC) is usually applied to compute repeated reachability probabilities.

Definition 2.24 (Strongly Connected Component (SCC))
Let be $\mathcal{A} = (Q, \Sigma, \delta, q_0)$ be a transition system. Two states q, and q' are mutually reachable iff $q \to q'$ and $q' \to q$. Mutual reachability is an equivalence relation on states of an LTS, and its equivalence classes are SCCs. An SCC is a terminal or bottom strongly connected component (BSCC) if no state of another SCC is reachable from any state of Q.

We observe SCC based probabilistic model checking in the literature as seen in the next related work chapter 3 and the details of the PCTL model checking and extensions can be found in [91] and [123].

2.4 SUMMARY

In this section, we provide a summary of foundations for the reader on model checking finite automata and regular expressions for a better understanding of the semantics of SREs over sequences that are constructed on traces and prefixes of the language. Therefore the first part includes the basics on regular languages and the relationship (conversions) between labeled transition systems and regular expressions as well as parsing regex and pattern matching.

Second part provides an overview of probability theory, randomness, and discrete event systems for the background of probabilistic behaviour of SRE language. We have also present the probabilistic model checking and its techniques for a clear understanding of the comparison between quantitative model checking of SRE model checking and conventional probabilistic model checking.

3 RELATED WORKS

Quantitative verification in the scope of software evolution has been much studied in the research of self-adaptive systems. The reason is that self-adaptive systems modify their behavior in response to changing environmental conditions, evolving requirements, and internal changes inside their feedback loop [106]. Therefore, *Efficient quantitative verification* (EQV) plays an important role in self-adaptive systems to enable them to comply with reliability, performance, and other QoS requirements throughout their lifetime.

Incremental quantitative verification is considered an EQV technique that avoids unnecessary computation by exploiting verification results from previous runs. Such a technique is applicable for self-adaptive systems since the changes are typically localized [20]. To this end, state-of-the-art incremental verification-based approaches are investigated in detail in the following section.

On the other hand, we explain the cast of common probabilistic models for quantitative verification in the upcoming section. This thesis presents a **new formalism** to propose an elegant and generic solution for incremental quantitative verification; thus, the related works adopt two distinct approaches: ① problem-related approaches for incremental probabilistic model checking, ② language-related approaches that have similar algebra to stochastic regular expressions.

3.1 PROBLEM-RELATED APPROACHES: SUPPORTING INCREMENTAL QUANTITATIVE VERIFICATION

We classify early approaches **supporting** incremental quantitative verification as *composition-based*, *abstraction refinement-based*, and *parametric verification* techniques in the following three subsections.

The fourth and final subsections discuss and compare the related works that focus solving the problem **directly** on the *incremental quantitative verification* techniques for the evaluation of evolving software. One can observe that the research on *incremental quantitative verification* has been highly investigated in very recent studies (see Table 1).

3.1.1 *Abstraction Refinement Based Approaches*

Abstraction, which hides the internal structure of the components and draws solutions to the state explosion problem, is a widely used technique in model checking.

Incremental behavior in abstraction algorithms naturally appears since the idea lies upon refining the abstract automata model through the guidance of counterexamples. The refinement algorithms iteratively use the intermediate runs until reaching a compromised abstract model [31, 114].

The probabilistic version of this approach was founded on traditional abstraction as well. Generating [78] and refining [83] probabilistic counterexamples are a key problem similar to the traditional abstraction refinement approaches.

The interesting point is that such technique is implemented in "extreme model checking" [80] in a way that the intermediate model computations are used for different model versions, namely changing models. Nevertheless, such a technique does not exist in a non-probabilistic setting to our knowledge.

3.1.2 *Compositional Verification in Probabilistic Models*

A second promising class of approaches for incremental evaluation is compositional techniques [55]. Even though this approach promises modularity of the models and localization of properties, it has not yet been exploited in an incremental fashion to our knowledge. One of the solutions to the compositional view for probabilistic model checking is assume-guarantee techniques [46, 101]. Compositional probabilistic verification decomposes components into separate subtasks for each system component like in the traditional composition in software verification; in particular, assume-guarantee verification techniques for MDPs [101] have recently been proved to provide real practical gains in scalability. The composition is also applied for interactive processes in [81].

Another study has proposed a framework that performs assume-guarantee reasoning on probabilistic automata in a fully automatic fashion based on the assume-guarantee techniques for standard verification [101].

3.1.3 *Parametric Model Checking*

The idea behind parametric probabilistic model checking lies in the uncertainty of probabilities at design time. Such uncertainty is denoted as parametric variables of transitions probabilities in a Markov chain. These variables can be calculated as rational functions that enable an update of probabilistic information at runtime. Therefore, parametric model checking can be counted as a way enabling the **reusability** of model checking results for **changing** probabilities of models.

An early attempt and application of symbolic and parametric model checking (PMC) to discrete-time Markov chain analysis can be seen in [36]. Although PMC focuses initially uncertainty, it encapsulates the problem of changing transition probabilities at run-time, and the implementation of the ideas is quite related to our approach. Therefore, we summarize the evolution of the parametric model checking in detail within the following studies.

Daws [36] introduces a technique that considers transition probabilities to be the letters of an alphabet of an FSM. Thus, the DTMC model is first transformed into an FSM that has the letters encoding the parametric probabilities. The probability measure of a set of paths satisfying a formula is computed symbolically as a regular expression on that alphabet, with the standard algorithms (state elimination [87]) to obtain a regular expression from an FSM. The regular expression is then evaluated to its exact rational value when the transition probabilities are rational. We clarify the approach with an example as follows in Figure 12 where transition probabilities are designed as letters of the FSM.

Hahn et al. [77] improve Daws' study by optimizing the algorithm during the state elimination method. The paper deals with the well-known "size explosion of regular expression" problem [69] applied in PMC. To curb this problem, **Hahn et al.** [76] pro-

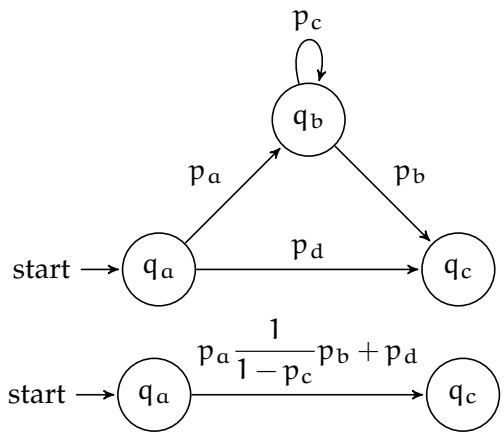


Figure 12: State elimination of parametric Markov chain [36]

vides an algorithm that calculates the rational function on the fly; such that it directly sifts letters not at the end of the state elimination, rather during the state elimination. Apart from that, they also consider extensions with rewards or non-determinism and describes the bisimulation relations for parametric Markov models.

Filieri et al. [48] present an alternative approach to incremental verification based on parametric model checking by first calculating the closed formula at design time once, and evaluating them at runtime afterwards. The computation of PCTL formulae is symbolically achieved with Gaussian elimination on the transition matrix of the parametric Markov chain. The obtained closed formula is then calculated at runtime efficiently. However, when the matrix changes, the closed formula is recalculated, which leads a high computation cost whenever the structure of the model changes.

Yamilet et al. [110] deals with the uncertainty with the perturbation approach. In the existence of the perturbation range, an asymptomatic analysis is performed on the stationary distribution of the DTMC.

A domain specific framework for robots in adaptive environments is presented recently by **Zhao et al.** [148] to verify the safety and reliability requirements of the robots, both at design stage and runtime. The approach introduces new estimators based on conventional Bayesian inference and imprecise probability model with sets of priors to learn the unknown transition parameters from operational profile applying the approach in unmanned underwater vehicle.

All these PMC solutions have problems with structural changes in the model since they are basically focusing on the parametric changes, which are transition probabilities. In other words, these techniques achieved efficient model checking effort, especially, for parameter changes and additional transitions of existing states.

3.1.4 *Incremental Probabilistic Verification*

Gainer et al. [53] has recently introduced the *volatile parametric Markov chain* that has volatile states, which configure the state elimination ordering for reachability property to provide incremental verification of parametric and reconfigurable Markov chains. Such reconfiguration provides high efficiency, especially, for recurrent state patterns like Zeroconf protocol [17] when a new state is added. An example of the zeroconf protocol model is demonstrated in Figure 13 showing one "state addition" to the Markov

chain. Such an approach provides the efficiency, however, it is very limited to "state addition" for recurring systems like Zeroconf protocol.

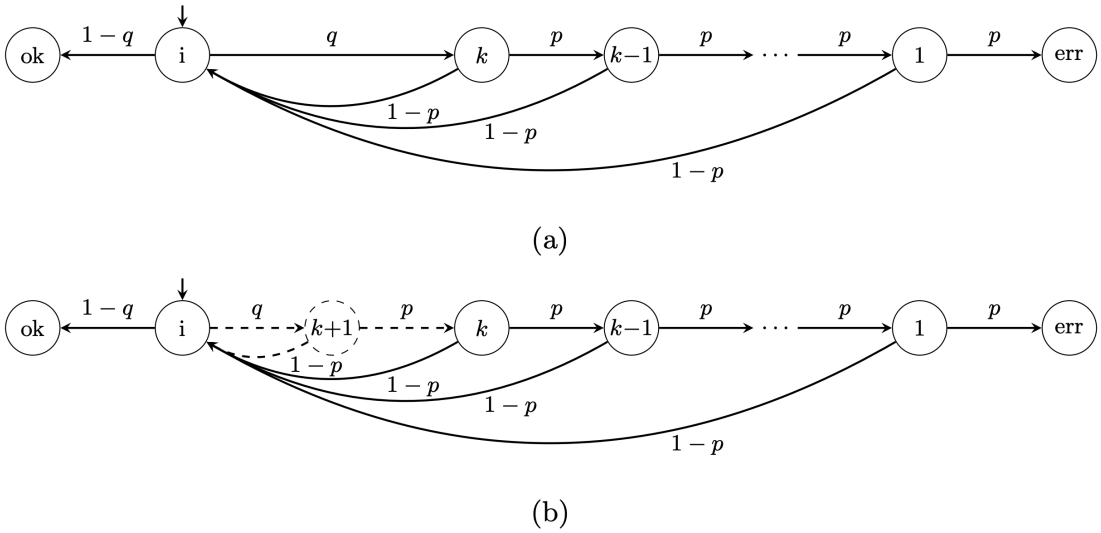


Figure 13: The Zeroconf protocol for $n = k(a)$ and $n = k + 1$ (b) [53]

Kwiatkowska et al. [102] has achieved an approach for incremental verification of changing models by using decomposition of Markov chains into strongly connected components. Decomposition of Markov decision processes into SCCs (Strongly connected components) and SCC based value iteration for quantitative verification has been introduced firstly in [29]. Incremental verification for parametric changes is enhanced with the decomposition of MDPs into strongly connected components.

An incremental approach for Markov decision based on the same techniques has been improved by eliminating the pre-computations and applying parallel computation [102]. In this approach, an exact analysis is performed with numerical computations based on the decomposition of Markov decision processes into SCCs with an optimized Tarjan algorithm [136] to exploit the model structure. After the first iteration is performed, when some parameters undergo changes, the probabilities are updated only for the affected states in the previous iteration. This technique is limited to parameter changes that are restricted in SCC [52].

Liu et al. [109] have recently introduced a heuristics-based approach that reduces the number of numerical iterations in the verification process of DTMCs by reordering the state of the model after the model changes, hence it could improve the runtime probabilistic model checking performance for some recurring cases in the model as it exists in the Crowds protocol.

Meedeniya et al. [116] introduces an algebraic delta evaluation that stores the computation and reuses the results only under a single parametric change.

			Model Type		
			DTMC	MDP	Algebraic
Incrementality for Adaptability	Parametric MC	Symbolic	Daws, 2005 [36] Hahn, 2009 [76] Filieri et al., 2011 [48]	Hahn et al., 2011 [77]	
		Decompositional		Kwiatkowska el al., 2011 [100] Forejt el al., 2012 [51]	
		Config. based		Gainer et al., 2018 [53] Liu et al., 2020 [109]	
	Component based		Calinescu et al., 2018 [24] Gerasimou et al., 2018 [56]		Johnson et al., 2013 [89]
Incrementality for Efficiency	Parametric MC		Filieri et al., 2011 [48]		Meedeniya et al., 2010 [116]
	Abstraction Refinement		Hahn et al., 2009 [76]	Hermanns et al., 2008 [83] Kattenbelt et al., 2010 [92]	
	Assume-Guarantee			Kwiatkowska et al., 2010 [104]	

Table 1: Classification of related quantitative verification approaches

Bianculli et al. [15] have applied a syntax driven approach for the reliability analysis of structured programs, where the general idea of a syntactic-semantic framework called SiDECAR is presented in [16]. The incremental verification framework uses operator precedence grammar together with attribute grammars to identify non-terminal nodes where the change occurs. Hence, the affected part of the abstract syntax tree can be replaced. This framework lies upon a very similar idea to the incremental quantitative verification framework (IQon) introduced in this thesis. Yet, SiDECAR is too generic, and addresses any type of model expressed in operator precedence grammar and exhaustively searches the parent node to apply the change, which might require quite amount of time for some cases.

On the other hand, IQon on Stochastic regular expression language captures probabilistic model checking, and remains modular for change operations leading time efficiency (A detailed comparison of SiDECAR and IQon is provided in Chapter 6).

3.1.5 *Configuration Based Approaches*

Johnson et al. [89] present an incremental evaluation framework with high-level algebraic representations of component-based systems to identify, and execute the re-verification for only the minimal set of components after a system change (component-wise addition, deletion, and modifications).

Calinescu et al. [24] propose a framework that calculates the closed form of the PCTL formulas [48] for the domain-specific modeling patterns. Firstly, the domain specific stage is performed only once, which requires domain-expert input to identify modeling patterns for the components of systems from the considered domain, and pre-computes closed-form expressions for key QoS properties of these patterns. Secondly, parametric model checking is performed for each structurally different variant of a system and the QoS property of a parametric Markov chain that models the interactions between the system components. In this framework, the Markov chain is modeled as a high-level model and configured based on the patterns.

Finally, **Ulusoy et al.** [140] present an incremental approach to synthesizing Markov decision processes of agent-based systems with standard verification techniques (LTL based model checking).

3.2 LANGUAGE RELATED APPROACHES: PROBABILISTIC ALGEBRAIC EXPRESSIONS

Several studies have been conducted to describe regular expressions in stochastic settings and Kleene-like languages [96].

Bollig et al. [19] introduce probabilistic regular expressions (PRE) and probabilistic *pepple expressions*. PREs are a version of weighted regular expressions and stochastic regular expressions. Unlike SREs, they are not encoded with the number of occurrences for choice terms, instead, the probabilistic distributions are attached to the choice term in PREs. More precisely, the representation of the term $E = a[4] + b[6]$ in SRE will be $E = a\frac{4}{10} + b\frac{6}{10}$ in PRE (The difference is only syntactic).

The paper establishes the equivalence between probabilistic pepple expressions and probabilistic *pepple automata*. The focus application of this paper is quantitative reason-

ing on database queries. An analysis of the same language for the temporal logic is also presented in **Monmege's thesis** [117] comprehensively.

Kartzow and Weidner [90, 143] define a probabilistic Monadic Second-Order Logic (MSOL), and a constraint logic with temporal properties for data analysis of probabilistic regular expressions. Additionally, Weidner specifies probabilistic regular expressions for infinite strings with ω-properties in their study [142].

McIver et al. [113] provide a framework for *probabilistic concurrent Kleene algebra* based on rely-guarantee rules and described in concurrent Kleene algebra [85] such that it describes the simulation and parallel composition for Kleene star. This framework is highly relevant to an extension of stochastic regular expressions and their bisimulation relations.

Table 2 summarises the languages discussed in this section. Even though the languages are very similar, we are interested in model checking against a probabilistic computational tree. Hence, we present the languages and the logic studied in the papers. In general, the classified languages in Table 2 are generic, nevertheless initial intention for the application domains is also provided. For example, our solution is targeting software models, but can also be applied to any application (such as speech recognition, neural networks, etc.); that is why we have "+" for other extensible applications.

	Language	Studied Logic	MC. alg	App. domain	Tool
This thesis	Stoc. Reg. Ex (Trees)	Prob. ACTL	yes	software models+	yes
Weidner et. al, 2016 [142]	Prob. Reg. Ex Trees	MSO	no	*	no
Monmege, 2013 [117]	Prob pepple lang.	Temp.L	no	image models+	yes
Bollig et. al, 2012 [19]	PRE & Prob.pepple lang	-	no	Query lang.	no
McIver et al., 2013 [113]	Prob concurrent Kleene algebra.	-	no	concurrent systems	no

Table 2: Kleene-like probabilistic languages in the context of SREs

Part II

QUANTITATIVE VERIFICATION OF STOCHASTIC REGULAR EXPRESSIONS

4 A NEW FORMALISM FOR QUANTITATIVE VERIFICATION

Probabilistic behavior and verification are essential for several application areas, such as software engineering, speech recognition, digital communications, and computational biology, among others. In the context of software engineering, probabilistic models are often developed as a means to express and assess quantitative requirements of a system, such as performance and reliability. These probabilistic models used for model checking can be at different levels of abstraction, such as Markov chains [9], Markov decision processes [9], and stochastic Petri nets [112].

On the other hand, regular expressions have spread through all areas of theoretical computer science and play an important role in the field of natural language processing, including parsing, deep language models, model inference, and machine translation.

In this chapter, however, we focus on stochastic regular expressions (SREs) in the model checking context and establish a semantics of stochastic regular expressions employing a probabilistic extension of Action-based Computation Tree Logic (ACTL) denoted as probabilistic action computational tree logic (PACTL) to reason about temporal properties quantitatively. Then, we present the model checking algorithm based on *pattern matching* over *translated SREs*, denoted as `propertySRE` from PACTL formula. Formally, our algorithm checks if an SRE E satisfies a `propertySRE` E_p ($E \models E_p$). Finally we demonstrate an application for the provided formalism and show the scalability of our approach.

In contrast to state-based representations in conventional probabilistic model checking, we introduce an approach focusing on stochastic regular expressions as an input model for probabilistic model checking applications. SREs are constructed as a tree and provide a means to localize probabilistic data inside the tree nodes. Thus, such **decomposition** of probabilistic data can be used for multiple problems, such as, component-based and iterative verification since it enables the **reusability** of the calculations for **changing** probabilistic systems [63].

4.1 FORMAL SEMANTICS FOR PROBABILISTIC VERIFICATION OF SRES

This section provides a foundation for the quantitative model checking of stochastic regular expressions (SREs) by defining the semantics over an action-based logic. Thus, the meaning of "satisfaction relation" for paths over SRE is manifested.

Stochastic Regular Expressions

The syntax of a SRE (E) is defined over an alphabet Σ recursively as follows [128]:

$$E := \alpha \left| \sum_i E_{i[n_i]} \right| \sum_i^G E_{i[n_i]} \left| E_1 : E_2 \right| E^{*f} \left| E^{+f} \right| (E) \tag{18}$$

with $\alpha \in \Sigma$, $n_i \in \mathbb{N}_0$, $f \in [0, 1] \subset \mathbb{R}$, and every term E_i is an SRE, such that:

1. *Atomic Action* α: α is an atomic action that belongs to the alphabet Σ.

2. *Choice* $\sum_i E_{i[n_i]}$: One of the provided terms is probabilistically chosen according to the given probabilities. n_i denotes the *occurrence value* for each term, such that the ith term is chosen with probability $\frac{n_i}{\sum_i n_i}$ (as an example for a term $E = a_{[5]} + b_{[3]}$, a and b are chosen with the probability $\frac{5}{5+3} \approx 0.625$ and $\frac{3}{5+3} \approx 0.375$, respectively). The *choice* operation is non-deterministic since the language allows the same action with different choice rates, e.g., $a_{[5]} + a_{[2]}$.

3. *Guarded Choice* $\sum_i{}^G E_{i[n_i]}$: Each sub-expression of choice operation has to be prefixed with a unique action or consist of a unique action by itself. That is, for $E = \sum_i{}^G E_{i[n_i]}$, e.g., $E = (\alpha_i : E_{1[n_1]} +^G \alpha_j : E_{2[n_2]})$ or $E_{i[n_i]} = \alpha_i$, and $\forall \alpha_i, \alpha_j : \alpha_i \neq \alpha_j$. This constraint makes *guarded choice* deterministic. Note that the calculation of probabilities is identical to the *choice* operation, and the *guarded choice* is a syntactic constraint in the design of the language. Moreover, the *guarded choice* does not remove non-determinism in the language completely, since the closure operators are non-deterministic.

4. *Concatenation* $E_1 : E_2$: The terms E_1 and E_2 are successively interpreted.

5. *Kleene Closure* E^{*f}: The term E is repeated for a potential number of times, subject to a binomial distribution. Each iteration occurs with a probability of f. The termination probability is $1 - f$ (for instance, $E = a^{*0.3}$ has the probability of 0.3 for each iteration, and 0.7 for the termination. In this case, E has to be executed twice and then terminate for the occurrence of the sequence "aa" with probability $0.3 \cdot 0.3 \cdot 0.7 = 0.063$).

6. *Plus Closure* E^{+f}: The term E executes at least once, and follows the same probability scheme as Kleene closure. Hence, *plus closure* is actually a syntactic sugar and can be easily emulated with $E : E^{*f}$.

7. *SRE in parentheses* (E): A syntactical addition to enable the operator precedence in desire (e.g., $(a_{[4]} + b_{[5]}) : c$). (E) is semantically equivalent to E. The parentheses have the highest precedence in the operations, and Kleene/plus closure, concatenation and choice follow it respectively.

A derivation of a conventional regular expression E is a set of sentences or strings over the alphabet. This derivation defines the language $L(E)$ of E. This notion of language derivation is similarly applicable to SRE, except that each string has a probability value associated with it; hence, the language itself is associated with a probability distribution of its members, as explained in the following subsection [128].

Mapping from Strings to Probability Space

Intuitively, the language of SREs can be considered that every expression E defines a function that $[\![E]\!] : s \in \Sigma^* \rightarrow [0, 1]$ where Σ^* is the infinite sets of the alphabet. Such a probabilistic language (p-language) is previously described for discrete events systems [99]; and it forms an algebra that is prefix-closed under the operations choice, concatenation and Kleene closure.

The semantics of SREs are described by the p-language in the style of denotational semantics [133]. The probability function for an SRE is denoted by $[\![E]\!]$, and its application to a particular string s is denoted as $[\![E]\!]s$, which indicates that an acceptance probability is associated with the string s in the language $L(E)$. The probability function calculates the occurrence probability of an arbitrary string $s \in \Sigma^*$ in an SRE model recursively. The language $L(E)$ is applied to the probabilistic pattern matching problem using genetic algorithms in [128]. This thesis scrutinizes the application and extension of this language to the model checking context. Let us explain the details of the ***probabilistic matching (occurrence)*** function based on the operations defined in the SRE syntax. The probability function $[\![E]\!]s$ for every possible SRE term in the syntax (18) is calculated recursively, where $s = \alpha_1 \dots \alpha_n \in \Sigma^*$ is a string and α_i is an arbitrary symbol [128].

- **Atomic actions:** For atomic action α is trivial. If the action equals to the string s (s is an action in this case), then the result is 1; otherwise 0.

$$\begin{aligned} [\![\alpha]\!]s &= 1, \quad \text{if} \quad \alpha = s \\ [\![\alpha]\!]s &= 0, \quad \text{if} \quad \alpha \neq s \end{aligned} \tag{19}$$

- **Choice:** Every term possibly recognize s; therefore, the overall probability for a choice expression is the sum of all the term probabilities with respect to s.

$$\left[\!\!\left[\sum_i E_{i[n_i]} \right]\!\!\right] s = \sum_k \left(\frac{n_k}{\sum n_k} \right) . [\![E_k]\!]s \tag{20}$$

- **Concatenation:** s is decomposed into two substrings in the first summation term, each of which may be consumed by a concatenated term. Although one term may recognize its substring argument, if the other term does not recognize its respective substring, then the result is zero probability. Eventually, the overall probability for that instance of decomposition is zero. The rest of the formula represents the cases when one entire expression consumes s, while the other consumes the ε. If those cases do not succeed, then the result is zero.

$$\begin{aligned} [\![E_1 : E_2]\!]s = \sum_{i=1}^{n} \left([\![E_1]\!]\alpha_1 \dots \alpha_i \cdot [\![E_2]\!]\alpha_{i+1} \dots \alpha_n \right) \\ + [\![E_1]\!]s \cdot [\![E_2]\!]\varepsilon \\ + [\![E_1]\!]\varepsilon \cdot [\![E_2]\!]s \end{aligned} \tag{21}$$

- **Kleene closure:** The first part of the formula accounts for the empty strings, meaning that the expression is not iterating (terminating without executing). The termination probability is, therefore, $1 - f$. The other part of the formula recursively defines the general case. More explicitly, one iteration of E will consume some portion of s, and the rest of s is consumed by further iterations.

$$\begin{aligned} [\![E^{*f}]\!]\varepsilon &= 1 - f \\ [\![E^{*f}]\!]s &= \sum_{i=1}^{n} \left(f \cdot [\![E]\!]\alpha_1 \dots \alpha_{i-1} \cdot [\![E^{*f}]\!]\alpha_i \dots \alpha_n \right) \\ &\quad + f \cdot [\![E]\!]s[\![E^{*f}]\!]\varepsilon \quad s \neq \varepsilon \end{aligned} \tag{22}$$

- **Plus Closure:** This case is similar to the non-empty argument formula for Kleene closure, except that the expression E consumes a part of the string at least once.

$$\begin{aligned}[\![E^{+f}]\!]s = \sum_{i=1}^{n} \left([\![E]\!]\alpha_1 \dots \alpha_i . [\![E^{*f}]\!]\alpha_{i+1} \dots \alpha_n\right) \\ + [\![E]\!]s \cdot [\![E^{*f}]\!]\varepsilon \text{ (If } E \text{ itself matches } s)\end{aligned} \tag{23}$$

Example 4.1 (Application of the probabilistic matching (occurrence) function) *Let an SRE* $E = \underbrace{(a:b)^{*0.35}{}_{[5]}}_{x} + \underbrace{a:b:d_{[10]}}_{y} + \underbrace{c_{[3]}}_{z} + \underbrace{(a:d)_{[2]}}_{t}$ *be defined on the alphabet* $\Sigma = \{a, b, c, d\}$.

What is the probability ab *matches* E *or element of* E*? That is* $[\![E]\!]ab$*. The matching probability of the string* ab *is calculated as follows using the probability function described above. The calculation starts with the inner part of every choice element.*

1. *For* $x = a : b$

$$\begin{aligned}[\![a:b]\!]ab &= [\![a]\!]a \cdot [\![b]\!]b \\ &+ [\![a]\!]\varepsilon \cdot [\![b]\!]ab \\ &+ [\![a]\!]ab \cdot [\![b]\!]\varepsilon \\ &= 1 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 = 1\end{aligned}$$

2. *The loop probability (Kleene star) comes into account:*

$$\begin{aligned}[\![(a:b)^{*0.35}]\!]ab &= 0.35 \cdot \overbrace{[\![a:b]\!]a}^{0} \cdot [\![(a:b)^{*0.35}]\!]b \\ &+ 0.35 \cdot [\![a:b]\!]ab \cdot \overbrace{[\![(a:b)^{*0.35}]\!]\varepsilon}^{①} \\ &= 0.35 \cdot 1 \cdot 0.65 = 0.2275\end{aligned}$$

3. *For* $y = a : b : d$*, we can reuse the results of* $x = a : b$*:*

$$\begin{aligned}[\![x:d]\!]ab &= \overbrace{[\![x]\!]a}^{②} \cdot [\![d]\!]b \\ &+ [\![x]\!]\varepsilon \cdot [\![d]\!]ab \\ &+ [\![x]\!]ab \cdot [\![d]\!]\varepsilon \\ &= 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 0 = 0\end{aligned}$$

① Termination probability is 0.65.

② We calculate $[\![a:b]\!]a$ similarly. Since the second part of $a : b$, which is b matches ε with zero; then the result is zero.

4. *The concatenation operation calculates every possible combination of* ab *string. Similarly,* z *and* t *give zero as a result. Finally, the probability of choice elements are taken.*

$$\begin{aligned}[\![E]\!]ab &= [\![x]\!]ab_{[5]} + [\![y]\!]ab_{[10]} + [\![z]\!]ab_{[3]} + [\![t]\!]ab_{[2]} \\ &= 0.2275 \cdot \frac{5}{20} + 0 \cdot \frac{10}{20} + 0 \cdot \frac{3}{20} + 0 \cdot \frac{2}{20} \\ &= 0.2275 \cdot 0.25 \approx 0.056\end{aligned}$$

Thus far, we have described the SRE language and the probabilistic function over strings. On the other side, our main goal is to reason about temporal properties probabilistically on the SRE models. A widely used logic probabilistic computation tree logic (PCTL) [79], and its variants are defined on the state and path formulas. However, stochastic regular expressions do not have an explicit notation of a *state*. Therefore, we prefer to extend the ACTL logic semantically expressed on the **runs** of the system, as provided in Chapter 2. A **run** of a system can be defined as a **word** in an SRE context. In other words, a word is an execution of a system, that is an element of the language, and it has a *length* in case of finiteness.

Definition 4.1 (Words of an SRE)
$\mathrm{Words}(E) = \{w \mid w \in L(E)\}$ *and* $w \in L(E)$ *iff* $[\![E]\!]w > 0$

Definition 4.2 (Word length)
The ***length*** *of a word* $w = w_0 w_1 \ldots w_n$ *is the number of included symbols and denoted as* $|w| = n$.

Based on these definitions, we have described the probabilistic extension of ACTL logic called PACTL in the following paragraph.

Definition 4.3 (Syntax of PACTL)
We define a logic as an extension of ACTL called Probabilistic ACTL (PACTL) and build the semantics over the SRE terms and the words of SREs.

$$\Phi = \mathtt{true} \mid \neg\Phi \mid \Phi \vee \Phi' \mid \Phi \wedge \Phi' \mid \mathcal{P}_P(\varphi) \tag{24}$$

$$\varphi = \mathcal{X}_a \Phi \mid \mathcal{X}\Phi \mid \Phi\, \mathcal{U}\, \Phi' \tag{25}$$

where $P \subseteq [0, 1]$ *is an arbitrary subset.*

The extension of the ACTL logic is mainly provided by the $\mathcal{P}_P(\varphi)$ operator and make a separation of *SRE* and *SRE word* formula. The recursive formulation of the syntax allows the nested $\mathcal{P}_P(\varphi)$ operator, although such formule are not common in practice, as presented in ProProSt Patterns [7, 71].

Definition 4.4 (Syntax of PACTL)
The semantics is defined as a satisfaction relation for an SRE term E *and a word* w *as follows [60]:*

$$E \models \neg\Phi \textit{ iff } E \not\models \Phi$$
$$E \models \Phi \vee \Phi' \textit{ iff } E \models \Phi \vee E \models \Phi'$$
$$E \models \Phi \wedge \Phi' \textit{ iff } E \models \Phi \wedge E \models \Phi'$$
$$E \models \mathcal{P}_P(\varphi) \textit{ iff } \Pr\{w \models \varphi \textit{ such that } w \in \mathtt{Words}(E)\} \in P$$
$$w \models \mathcal{X}_a\Phi \textit{ iff } w[0] = a \wedge w[1] \models \Phi$$
$$w \models \mathcal{X}\Phi \textit{ iff } w[1] \models \Phi$$
$$w \models \Phi\,\mathcal{U}\,\Phi' \textit{ iff for some } i < |w|,\ w[i] \models \Phi' \wedge w[j] \models \Phi, \forall j < i.$$

The negation, conjunction, and disjunction are applied on the SRE term E after the word formulae are calculated. The word formulae are constructed on the temporal properties such as *Next* ($\mathcal{X}$) and *Until* ($\mathcal{U}$). A word with index refers to an exact location of a symbol in the word, meaning that $w[i]$ represents the symbol at location i. Furthermore, $w[i] \ldots w[k]$ stands for the symbols from the i*th* symbol until k*th* symbol.

The measure $\Pr\{w \models \varphi$ such that $w \in \mathtt{Words}(E)\}$ corresponds to the probabilistic matching function of w denoted as $[\![E]\!]w$.

Put another way, the calculation of the probabilistic operator is formed with the probabilistic value of the words. The probability function of an SRE term $[\![E]\!]s$ for a string s is a mapping from Σ^* to a probabilistic value $p \in [0, 1]$. Let us provide an example to see the string sets, with their probabilities that are calculated by the probability function.

Example 4.2 (Word sets of SREs)
Let E_1, E_2, E_3 *SREs be defined on the alphabet* $\Sigma = \{\varepsilon, a, b, c\}$*, then:*

1. $E_1 = a : b$ *and*
   the matching probability of string "ab" is 1.0 then,
   $\rightarrow$ *string set of* $E_1 = \{ab(1.0)\}$.

2. $E_2 = a : b_{[4]} + c_{[6]}$ *then, the matching probability of string "ab" is* $\frac{4}{4+6} = 0.4$ *for string ' 'c" is* $\frac{6}{4+6} = 0.6$ *then,*
   $\rightarrow$ *string set of* $E_2 = \{ab(0.4), c(0.6)\}$.

3. $E_3 = (ab_{[4]} + c_{[6]})^{*0.3}$*, and we consider the termination probability* 0.7.

   - *the matching probability of string "ab" is* $\overbrace{0.4}^{\frac{4}{4+6}} \cdot 0.3 \cdot 0.7 = 0.084$,
   - *the matching probability of string "abab"* $0.4 \cdot (0.3)^2 \cdot 0.7 = 0.0252$,
   - *the matching probability of string "c" is* $0.6 \cdot 0.3 \cdot 0.7 = 0.126$*, and*
   - *the matching probability of string "cab" is* $0.6 \cdot 0.3 \cdot 0.4 \cdot 0.3 \cdot 0.7 = 0,01512$ *then,*

   $\rightarrow$ *string set of* E_3 *is infinite so that* $E_3 = \{\varepsilon(0.7), ab(0.084), abab(0.0252), c(0.126), cab(0.01512), \ldots\}$.

4.2 MODEL CHECKING USING PATTERN MATCHING

In the conventional model checking approaches, the paths of the model M are first computed, and then checked, if they satisfy the given property P. The fundamental idea of model checking linear properties is to reason if the language of a model is included in the language of the given property. Formally, if $M \models P$, then $Words(M) \subseteq Words(P)$. The idea behind the model checking approach in this paper is similar yet in another perspective such that *the language inclusion problem is verbalized as a pattern matching problem* of SREs against a given property (satisfaction relation $\rightarrow$ language inclusion $\rightarrow$ pattern matching).

For this reason, we provide a translation from the PACTL property P into the SRE language, denoted as `propertySRE` (E_P), to check whether $E \models E_P$.

In the literature, the problem of whether $E \models E_P$ can also be seen as the "Implication of regular expressions." According to the definition in [137], a string satisfies a regex property ($w \models E_P$) in the following condition:

1. $w \models E_P$ if there exists a substring w' of w such that $w' \in L(E_P)$.
2. $E \models E_P$ if $w \models E_P$ whenever $w \models E$.

In model checking SREs, the first definition for `propertySRE` is intuitively similar to the definition of $w \models$`propertySRE` such that $w \models E_P$ holds according to the semantics given in (27). The second definition, however, is different because PACTL is not a linear logic; instead, it considers also the probabilistic branches (i.e., a type of branching-time logic). More explicitly, the probability of $E \models E_P$ covers all possible branches $E_1 \dots E_n$ of E to appear. Hence, each probability of $E_i \models E_P$ has to be summed up to the probability of $E \models E_P$.

In the following subsection 4.2.1, the translation of the PACTL formulas into SREs is provided. Then, the computation of the probabilities of *Flat Next*, *Flat Until*, and *Until* formulae (as bounded and unbounded reachability) are presented according to the translation in the subsections 4.2.1.1, 4.2.1.2, 4.2.2.1 and 4.2.2.2, respectively. Finally, a generalized algorithm of SRE model checking is demonstrated in section 4.2.3.

4.2.1 *Translating PACTL Word Formulas into SREs*

We implement the idea of model checking PACTL against SRE by translating every PACTL *word formula* into a stochastic regular expression. We borrow some ideas from *Regular Linear Temporal Logic* during the translation [107]. Overall, regular linear temporal logic differs from our approach in two ways: (1) we analyze the set of finite words in branches rather than ω words, and (2) we use measures that map from a string to a probability value. The translation function t : PACTL word formulas $\rightarrow$ SRE term is as follows:

$$t(\varphi) = \begin{cases} \textbf{true} \rightarrow \Sigma^* \\ \mathcal{X}_a \Phi \rightarrow a : t(\Phi) \\ \mathcal{X}\Phi \rightarrow \textbf{true} : t(\Phi) \\ \Phi \, \mathcal{U} \, \Phi' \rightarrow t(\Phi)^* : t(\Phi') \end{cases} \tag{26}$$

The E term formulas such as conjunction and disjunction are not directly applicable to a stateless logic like PACTL. For instance, one can check if a state satisfies the propositions "a" and "b" at the same time. Hence, this case is irrelevant for an action-based logic whose semantics is defined on its *runs*, and only one immediate action can be checked at one time. Instead, one can model check if a conjunction or a disjunction of the results, which are computed by the satisfaction relation of probabilities for different actions, is correct. Therefore, we can translate only word formulas into SREs (i.e., regex as a subset of SREs).

Let x and y be stochastic regular expressions translated from PACTL, and w is a word, then words of an SRE should satisfy the translated regular expressions recursively as follows:

$$
\begin{aligned}
&w \models a \text{ iff } a \text{ is prefix for the word } w.\\
&w \models x : y \text{ iff } w[0]...w[k] \models x,\ w[k+1]...w[j] \models y, \text{ and } |w| = j+1 \text{ for some index } k. \quad (27)\\
&w \models x^*y \text{ iff } w[i] \models y \text{ and } w[j] \models x \text{ for some } i \leqslant |w|, \text{ where } j = 0,.., i-1.
\end{aligned}
$$

The semantic style used above is similar to the temporal logic and associates a language over finite words to a given expression. In this manner, the definition of the Kleene star (x^*y) is equivalent to the conventional definition, that is
$L(x^*y) = L(y + xy + xxy + \ldots) = L(\sum_{i \geqslant 0} x^i : y)$ [137].

Translated PACTL properties as an SRE symbolize finite or infinite sets of words. In the following subsection, we explain the calculation for probabilistic matching of SREs against each translated SRE based on the translation function $t(\varphi)$. Firstly, the computations are provided for the flat formulas, which are not nested. Then, we convey the generalized algorithm.

Example 4.3 (Nested PACTL formula)
Let $\mathcal{P}_{[0.9,1.0]}(\mathcal{P}_{[0,1]}(\mathcal{X}_a(\textbf{\textit{true}})))\ \mathcal{U}\ (\mathcal{P}_{[0.01,0.5]}(\mathcal{X}_b(\textbf{\textit{true}})))\ \vee\ \mathcal{P}_{[0,1]}(\mathcal{X}_c(\textbf{\textit{true}})$ *be a PACTL formula. The model checking process of* $E \models E_p$ *starts with parsing the PACTL formula whose syntax tree is demonstrated in Figure 14. Once the PACTL formula is translated into* `propertySRE` E_p *and model checked then, the result is propagated bottom up until the whole formula is checked. For instance, the formula* $\mathcal{X}_a(\textbf{\textit{true}})$ *inside the dotted box is translated into* $a : (\mathrm{true})^*$*; hence, the formula* $\mathcal{P}_{[0,1]}(a : (\mathrm{true})^*)$ *is checked. The result (true or false) is directly substituted in the node. Eventually, the whole formula is checked by the propagation of the computed results.*

Translated PACTL properties as an SRE symbolize finite or infinite sets of words. In the following subsection, we explain the calculation for probabilistic matching of each translated SRE based on the translation function $t(\varphi)$. Firstly, the computations are provided for the flat formulas that are not nested. Afterwards we convey the generalized algorithm.

4.2.1.1 *Flat Next Formula*

The flat form (not nested) of the *next* formula $\mathcal{X}_a\Phi$, where Φ is **true**, corresponds to $a : s$ as an SRE, where $s \in \Sigma^*$ is an arbitrary string. In other words, all strings that have the specific a as the symbol in the first location satisfy the formula $\mathcal{X}_a(\textbf{true})$. Therefore

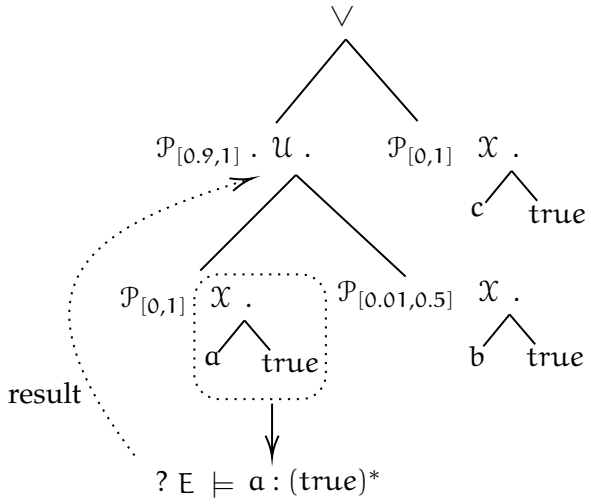


Figure 14: Parsing nested PACTL formula

searching string sets in the form of $a : \alpha$ is interpreted that "a" is a *prefix* and the rest is any word form.

We note that the next formula does not check the next state (differently from PCTL) but, the immediate action that changes the state to the next one in the system. Therefore, $\mathcal{P}_P(\mathcal{X}_a\Phi)$ scrutinizes whether the probability of a being *prefix* is satisfied with bound P for the given E. The ultimate goal is to check if E has some words that include "a" as an action in the first position and what the probability of that case is.

The probability function provided above (functions from (19) to (23) in Subsection 4.1) is useful when a whole path is checked whether it is an observation of the system. On the other hand, such a function is inadequate to reason about probabilistic properties like $\mathcal{X}$, $\mathcal{U}$, etc. Therefore, we define novel probabilistic functions recursively, such as *prefix*, *suffix*, and *infix*, defined on the top of the probability function using the algebraic features of SREs.

Prefix probability of a string for an SRE denoted as $[\![E]\!]_{\text{prefix}}s$, which is a means to reason the next formula, is recursively defined as follows:

$$[\![E]\!]_{\text{prefix}}s = \begin{cases} 1, & \text{if } E = \alpha \wedge s = \alpha \\ 0, & \text{if } E = \alpha \wedge s \neq \alpha \\ \sum_k \left(\frac{n_k}{\sum_j n_j}\right) \cdot [\![E_k]\!]_{\text{prefix}}s, & \text{if } E = \sum_i E_i(n_i) \\ \sum_{i=1}^{n} \left([\![E_1]\!]\alpha_1 ... \alpha_i \cdot [\![E_2]\!]_{\text{prefix}}\alpha_{i+1} ... \alpha_n\right), & \text{if } E = E1 : E2 \\ \sum_{i=1}^{n} \left(f \cdot [\![E]\!]\alpha_1 ... \alpha_i) \cdot [\![E^{*f}]\!]_{\text{prefix}}\alpha_{i+1} ... \alpha_n\right), & \text{if } E = E^{*f} \end{cases} \tag{28}$$

A detailed description of (28) is as follows:

- **Atomic actions:** Trivially, atomic actions are *prefixes* for themselves with probability 1; otherwise 0. The probability value for exact matching of ε is 1, if and only

if the node E is empty. However, ε is a *prefix* for any node E; and therefore, the *prefix* probability for ε is always 1.

$$[\![E]\!]_{\text{suffix}}\varepsilon = 1, \quad \text{for any } E \tag{29}$$

$$[\![\alpha]\!]_{\text{prefix}}s = 1, \quad \text{if} \quad \alpha = s$$

$$[\![\alpha]\!]_{\text{prefix}}s = 0, \quad \text{if} \quad \alpha \neq s \tag{30}$$

- **Choice:** Since every term might (disjointly) recognize s as a *prefix*, the overall probability for a choice expression is the sum of all the term probabilities with respect to s.

$$\left[\!\!\left[\sum E_i(n_i)\right]\!\!\right]_{\text{prefix}} s = \sum_k \left(\frac{n_k}{\sum n_k}\right) \cdot [\![E_k]\!]_{\text{prefix}}s \tag{31}$$

- **Concatenation:** We consider all the possible composition of string s and get the summation of them. In the first summation, s is decomposed into two substrings (s_1 and s_2). Once the first term recognizes its substring argument (s_1 exactly matches to E_1), the latter term (E_2) is supposed to have its respective substring s_2 as a *prefix*; otherwise, that term returns a probability of zero, and the overall probability for that instance of decomposition is zero. The rest of the formula represents the case when E_1 includes the entire string as a *prefix*, or E_1 matches only with ε where E_2 has the entire string as a *prefix*. If these cases do not succeed, then the result is zero. As a matter of fact, such a *prefix* formula recognizes the words of E. In other words, if a string s is a word of E, then it is also a *prefix* of E. The reversed case is not always true.

$$\begin{aligned}[\![E_1 : E_2]\!]_{\text{prefix}}s = \sum_{i=1}^{n} \left([\![E_1]\!]\alpha_1 \ldots \alpha_i \cdot [\![E_2]\!]_{\text{prefix}}\alpha_{i+1} \ldots \alpha_{n-1}\right) \\ +[\![E_1]\!]s \cdot [\![E_2]\!]_{\text{prefix}}\varepsilon \\ +[\![E_1]\!]\varepsilon \cdot [\![E_2]\!]_{\text{prefix}}s \quad s \neq \varepsilon \\ \cup[\![E_1]\!]_{\text{prefix}}s\end{aligned} \tag{32}$$

- **Kleene closure:** The matching probability for an empty string (the termination probability without executing) is defined as $1 - f$. However, the *prefix* probability of ε is always 1 since it is assumed that every word starts with ε. The formula is recursively defined as concatenation of two terms. Here, one iteration of E will consume some portion of s, and the rest of s is consumed as a *prefix* by further iterations. The final term in this formula represents the case that the first iteration consumes the entire string.

$$\begin{aligned}[\![E^{*f}]\!]_{\text{prefix}}s = \sum_{i=1}^{n} \left(f \cdot [\![E]\!]\alpha_1 \ldots \alpha_i \cdot [\![E^{*f}]\!]_{\text{prefix}}\alpha_{i+1} \ldots \alpha_{n_1}\right) \\ \cup f \cdot [\![E]\!]_{\text{prefix}}s\end{aligned} \tag{33}$$

Plus closure is similar to the non-empty argument formula for *Kleene closure*, except that the expression E will be executed at least once.

Example 4.4 (Calculation of the *prefix* probability)
Let an SRE $E = a_{[50]} + (d : a)_{[90]} + (a : b : c)_{[60]}$, *which is defined on the alphabet* $\Sigma = \{a, b, c, d\}$. *The probability of* a *being prefix for SRE* E *is calculated as follows:*

1. *For* a*, it is trivial.*

$$[\![a]\!]_{prefix} a = 1.$$

2. *For* $d : a$

$$\begin{aligned}[\![d : a]\!]_{prefix} a &= [\![d]\!] a \cdot [\![a]\!]_{prefix} \varepsilon \\ &= 0 \cdot 1 = 0\end{aligned}$$

3. *For* $a : b : c$*,* $[\![a : b]\!]_{prefix}$ *is firstly calculated.*

$$\begin{aligned}[\![a : b]\!]_{prefix} a &= \overbrace{\left([\![a]\!] a \cdot [\![b]\!]_{prefix} \varepsilon + [\![a]\!] \varepsilon \cdot [\![b]\!]_{prefix} a\right)}^{x} + [\![a]\!]_{prefix} a - x \cdot [\![a]\!]_{prefix} a \\ &= 1 \cdot 1 + 0 \cdot 0 + 1 - 1 \cdot 0 = 1\end{aligned}$$

4. *For* $(a : b) : c$

$$\begin{aligned}[\![a : b : c]\!]_{prefix} a &= \overbrace{\left([\![a : b]\!] a \cdot [\![c]\!]_{prefix} \varepsilon\right)}^{y} + [\![a : b]\!]_{prefix} a - y \cdot [\![a : b]\!]_{prefix} a \\ &= 0 + 1 - 0 \cdot 1 = 1\end{aligned}$$

5. *Finally, the concatenation operation calculates every possible combination of* a *as a string.*

$$\begin{aligned}[\![E]\!]_{prefix} a &= [\![a]\!]_{prefix} a_{[50]} + [\![d : a]\!]_{prefix} a_{[90]} + [\![a : b : c]\!]_{prefix} a_{[60]} \\ &= 1 \cdot \frac{50}{200} + 0 \cdot \frac{90}{200} + 1 \cdot \frac{60}{200} \\ &= 0.25 + 0 + 0.30 = 0.55\end{aligned}$$

4.2.1.2 *Flat Until Formula*

The plain version of the flat (not nested) *Until*($\mathcal{U}$) is represented as "$\mathcal{X}_a(\textbf{true})\, \mathcal{U}\, \mathcal{X}_b(\textbf{true})$" in our logic, which simply means $a\, \mathcal{U}\, b$.

In other words, the formula $\mathcal{X}_a(\textbf{true})\, \mathcal{U}\, \mathcal{X}_b(\textbf{true})$ searches the infinite sequences of the words that have the series "ab,aab,aaab..." as *prefixes*. The probability of satisfying the formula $\mathcal{P}(\mathcal{X}_a(\textbf{true})\, \mathcal{U}\, \mathcal{X}_b(\textbf{true}))$ is then, the *prefix* probability of the provided SRE term E:

$$[\![E]\!]_{\text{prefix}} a^{*f} b = [\![E]\!]_{\text{prefix}} ab + [\![E]\!]_{\text{prefix}} aab + [\![E]\!]_{\text{prefix}} aaab \ldots \tag{34}$$

$$[\![E]\!]_{\text{prefix}} a^{*f} b = [\![E]\!]_{\text{prefix}} ab + f \cdot [\![E]\!]_{\text{prefix}} ab + f^2 \cdot [\![E]\!]_{\text{prefix}} ab \ldots \tag{35}$$

The limit of this sum will be the geometric series of the *prefix* function:

$$f \cdot \frac{[\![E]\!]_{\text{prefix}} ab}{1 - f} \tag{36}$$

Note that the *prefix function* above, is calculating the probability of **exactly** "a" being *prefix* and not including "aa", "aaa"... being *prefix*. Therefore we extend the *prefix function* ($[\![E]\!]_{\text{prefix}} ab$) for the Kleene case by dividing $1 - f$, which means the *prefix* probabilities of a, is the max probability and includes all "a", "aa", ... (a^*). In other words, the maximum probability will be the *prefix* probability of a^* being *prefix*.

4.2.2 *Reachability Analysis*

The reachability analysis is applied on an SRE with a similar reasoning as in the discrete-time Markov chain (DTMC) reachability analysis. Unlike DTMCs, the goal is to calculate the probability of reaching a sequence of actions instead of a state label or labels. In the conventional probabilistic reachability analysis, the states satisfying the labels are identified, and the model checking is applied between these states. Nevertheless, in string analysis, we check if the action (sequence) is included in the words of the given SRE term locally, which does not require any graph traversal for the reachability. The grounding reason for that situation is that SREs are a different abstraction of models. We developed the reachability analysis by using such functions based on the string search. We discuss the details in case of bounded and unbounded reachability analysis in the following subsections.

4.2.2.1 *Unbounded Probabilistic Reachability*

The unbounded reachability analysis is performed by verifying the property **true** "*Until*" up to reaching a target action or action sequence. The plain formula of reaching an action "a" in PACTL is encoded as $(\mathbf{true}^*)\,\mathcal{U}\mathcal{X}_a(\mathbf{true})$.

The other way of expressing reaching an action a is that all the *prefix* strings, like "$\neg aa$", "$\neg a\neg aa$", "$\neg a\neg a\neg aa\ldots$", satisfy the reachability property. More explicitly, the question is "What is the probability of including the action of "a" in a given sequence?" We can also analyze and generalize the search as a sequence of actions like "$a_1a_2\ldots a_n$" in the formula $(\mathbf{true}^*)\,\mathcal{U}(\mathcal{X}_{a_1a_2\ldots a_n}(\mathbf{true}))$.

The reachability of actions in a set of infinite words for SRE means that the action set is an ***infix*** for any word of the SRE, which is included in any position of that word.

For instance, "a", "ab", "bc", "d," etc. are *infixes* for the word "abcd". Note that "a" is also a *prefix* and "d" is a *suffix*. The concatenation of "ab" and "cd" will be the word "abcd". "b" is a *suffix* for "ab", and "c" is a *prefix* for "d". Such information helps when we reason whether "bc" is included in the concatenated word "abcd".

In summary, we say that if s_1 is a *suffix* for any word w_1 and s_2 is a *prefix* for any word w_2, then s_1s_2 will be an *infix* for $w_1 : w_2$. We use that inference in the following calculations. Let us first provide the definitions of *suffix* and *infix* briefly. Assume that we search reaching an action sequence "ab" for a given SRE,

$$E = \underbrace{c : a : b_{[3]}}_{path_1} + \underbrace{d : a : b : c : d_{[8]}}_{path_2} + \underbrace{(b)^{*0.3}_{[5]}}_{path_3} \text{ on the alphabet } \Sigma = \{a, b, c, d\}.$$

One can observe that ab is reachable on $path_1$ and $path_2$. Unlike the reachability analysis in the conventional probabilistic model checking, we apply the **infix** search and calculate the probability by the *infix* function, as described in detail below.

The following function (37) is calculating if an SRE has a specific string ($s = \alpha_1\alpha_2\ldots\alpha_n$) as a *suffix*, which depends on the *matching function* (see function (4.1)).

$$[\![E]\!]_{\text{suffix}}s = \begin{cases} 1, & \text{if } E = \alpha \wedge s = \alpha \\ 0, & \text{if } E = \alpha \wedge s \neq \alpha \\ \sum_k \left(\frac{n_k}{\sum_j n_j}\right) \cdot [\![E_k]\!]_{\text{suffix}}s, & \text{if } E = \sum_i E_i(n_i) \\ \sum_{i=1}^{n} \left([\![E_1]\!]_{\text{suffix}}\alpha_1..\alpha_i \cdot [\![E_2]\!]\alpha_{i+1}\ldots\alpha_n)\right), & \text{if } E = E1 : E2 \\ \sum_{i=1}^{n} \left(f \cdot [\![E]\!]_{\text{suffix}}\alpha_1\ldots\alpha_i \cdot [\![E^{*f}]\!]\alpha_{i+1}\ldots\alpha_n)\right), & \text{if } E = E^{*f} \end{cases} \tag{37}$$

- **Atomic actions:** Trivially, atomic actions are *suffixes* for themselves with probability 1; otherwise 0. The probability value for exact matching of ε is 1 if and only if the node E is empty. However, ε is also a *suffix* for any SRE E, therefore, the *suffix* probability is always 1 for ε.

$$\begin{aligned} [\![E]\!]_{\text{suffix}}\varepsilon &= 1, \quad \text{for any } E \\ [\![\alpha]\!]_{\text{suffix}}s &= 1, \quad \text{if} \quad \alpha = s \\ [\![\alpha]\!]_{\text{suffix}}s &= 0, \quad \text{if} \quad \alpha \neq s \end{aligned} \tag{38}$$

- **Choice:** Since every term might (disjointly) recognize s as a *suffix*, the overall probability for a choice expression is the sum of all the term probabilities with respect to s.

$$\left[\!\!\left[\sum E_i(n_i)\right]\!\!\right]_{\text{suffix}} s = \sum_k \left(\frac{n_k}{\sum n_k}\right) \cdot [\![E_k]\!]_{\text{suffix}}s \tag{39}$$

- **Concatenation:** Similarly to the *prefix* function (28), we consider all the possible composition of string s, and get the summation of them. In the first summation, s is decomposed into two substrings (s_1 and s_2). Once the first substring (s_1) is a *suffix* for the first term (E_1), and the latter term recognizes its substring argument (s_2 exactly matches to E_2). The rest of the formula represents the case when E_1 includes the entire string as *suffix* or E_1 matches only with ε where E_2 has the entire string as a *suffix*. If these cases do not succeed, then it returns zero. If a string s is a word of E, then it is also a *suffix* of E. The reversed case is not always true.

$$\begin{aligned} [\![E_1 : E_2]\!]_{\text{suffix}}s = \sum_{i=1}^{n} \left([\![E_1]\!]_{\text{suffix}}\alpha_1\ldots\alpha_i \cdot [\![E_2]\!]\alpha_{i+1}\ldots\alpha_{n-1}\right) \\ +[\![E_1]\!]_{\text{suffix}}s \cdot [\![E_2]\!]\varepsilon \\ +[\![E_1]\!]_{\text{suffix}}\varepsilon \cdot [\![E_2]\!]s \quad s \neq \varepsilon \\ \cup[\![E_1]\!]_{\text{suffix}}s \end{aligned} \tag{40}$$

- **Kleene closure:** The matching probability for an empty string (the termination probability without executing) is defined as $1 - f$. However, it is always 1 for *suffix* probability since it is assumed that every finite word ends with ε. The formula

is recursively defined as the concatenation of two terms. Here, one iteration of E will consume some portion of s, and the rest of s is consumed as a *suffix* by further iterations. The final term in this formula represents the case that the first iteration consumes the entire string as a *suffix*.

$$[\![E^{*f}]\!]_{\text{suffix}} s = \sum_{i=1}^{n} \left(f \cdot [\![E]\!]_{\text{suffix}} \alpha_1 \ldots \alpha_i \cdot [\![E^{*f}]\!] \alpha_{i+1} \ldots \alpha_{n_1} \right) \cup f \cdot [\![E]\!]_{\text{suffix}} s \quad s \neq \varepsilon \tag{41}$$

Using the *prefix* and *suffix* functions, we describe the following ***infix*** function:

$$[\![E]\!]_{\text{infix}} s = \begin{cases} 1, & \text{if } E = \alpha \wedge s = \alpha \\ 0, & \text{if } E = \alpha \wedge s \neq \alpha \\ \sum_k \left(\frac{n_k}{\sum_j n_j} \right) \cdot [\![E_k]\!]_{\text{infix}} s, & \text{if } E = \sum_i E_i(n_i) \\ \sum_{i=1}^{n} \left([\![E_1]\!]_{\text{suffix}} \alpha_1 \ldots \alpha_i \cdot [\![E_2]\!]_{\text{prefix}} \alpha_{i+1} \ldots \alpha_n \right. & \text{if } E = E1 : E2 \\ \sum_{i=1}^{n} \left(f \cdot [\![E]\!]_{\text{suffix}} \alpha_1 \ldots \alpha_i \cdot [\![E^{*f}]\!]_{\text{prefix}} s_{i+1} .. s_n \right. & \text{if } E = E^{*f} \end{cases} \tag{42}$$

- **Atomic actions:** Atomic actions include themselves with probability 1; otherwise it is 0. ε is a *prefix* and a *suffix* for any node E, and therefore, it is also an *infix*, so that the probability is always 1.

$$\begin{aligned} [\![\alpha]\!]_{\text{infix}} s &= 1, \quad \text{if} \quad \alpha = s \\ [\![\alpha]\!]_{\text{infix}} s &= 0, \quad \text{if} \quad \alpha \neq s \end{aligned} \tag{43}$$

- **Choice:** Since every term might (disjointly) include s, the overall probability for a choice expression is the sum of all the term probabilities with respect to s.

$$\left[\!\!\left[\sum E_i(n_i) \right]\!\!\right]_{\text{infix}} s = \sum_k \left(\frac{n_k}{\sum n_k} \right) \cdot [\![E_k]\!]_{\text{infix}} s \tag{44}$$

- **Concatenation**: We consider all the possible composition of string s and get the summation of them. In the first summation, s is decomposed into two substrings (s_1 and s_2). Once the substring argument of the first term is a *suffix* for itself (s_1 is a *suffix* for E_1), the latter term (E_2) is supposed to have its respective substring s_2 as a *prefix*, otherwise that term returns a probability of zero, and the overall probability for that instance of decomposition is zero. The rest of the formula represents the case either E_1 or E_2 includes the entire string as an *infix*. If the union of these cases ((45), (46) or (47)) do not succeed, then it returns zero.

$$[\![E_1 : E_2]\!]_{\text{infix}} s = \sum_{i=1}^{n} \left([\![E_1]\!]_{\text{suffix}} \alpha_1 \ldots \alpha_i \cdot [\![E_2]\!]_{\text{prefix}} \alpha_{i+1} \ldots \alpha_{n-1} \right) \tag{45}$$

$$\cup [\![E_1]\!]_{\text{infix}} s \tag{46}$$

$$\cup [\![E_2]\!]_{\text{infix}} s \tag{47}$$

- **Kleene closure**: The formula is recursively defined as in the concatenation of two terms. The first iteration of E will have some portion of s as a suffix, and the rest will have s as a *prefix* by further iterations (Note that we use the extended *prefix* formula given in 36).

$$[\![E^{*f}]\!]_{\text{infix}} s = \sum_{i=1}^{n} \left(f \cdot [\![E]\!]_{\text{suffix}} \alpha_1 \ldots \alpha_i \cdot [\![E^{*f}]\!]_{\text{prefix}} \alpha_{i+1} \ldots \alpha_{n_1} \right) \cup f \cdot [\![E]\!]_{\text{infix}} s \quad s \neq \varepsilon \tag{48}$$

The reachability algorithm (presented in Algorithm 1) is built on the *prefix*, *suffix*, and *infix* functions for every SRE term. A system is specified as an SRE tree (i.e., a syntax tree obtained by parsing SRE) that includes a root node and a finite set of nodes. An SRE tree is formally defined as $T_E = (E_{\text{root}}, \mathcal{E}, \Sigma)$, where E_{root} is the root node, $\mathcal{E}$ is the finite set of all nodes, and Σ is the alphabet. The *infix* probability is calculated for every SRE node by substitution up to the root node E_{root}.

```
Reachability(E, s)
Data: An SRE tree T_E, target action sequence s
Result: The reachability probability for a given action/action sequence s
foreach E ∈ ℰ do
    if E is an "action node" α then
        if α = s then
            [[E]]_{prefix,suffix,infix} s ← 1
        else
            [[E]]_{prefix,suffix,infix} s ← 0
    else if E is a "sum node", E = E_1[n_1] + E_2[n_2] then
        [[E]]_{prefix,suffix,infix} ← Σ_i ( n_i / Σ_j n_j ) · [[E_i]]_{prefix,suffix,infix} s
    else if E is a "concat node", E = E_1 : E_2 then
        [[E]]s ← Σ_{i=1}^{n} [[E_1]]s_1..s_i · [[E_2]]s_{i+1}..s_n
        [[E]]_prefix s ← Σ_{i=1}^{n} [[E_1]]s_1..s_i · [[E_2]]_prefix s_{i+1}..s_{n-1}
        [[E]]_suffix s ← Σ_{i=1}^{n} [[E_1]]_suffix s_1..s_i) · [[E_2]]s_{i+1}..s_n
        [[E]]_infix s ← Σ_{i=1}^{n} [[E_1]]_suffix s_1..s_i · [[E_2]]_prefix s_{i+1}..s_n
    else if E is a "Kleene node", E = E^{*f} then
        Σ_{i=1}^{k} [[E]]s ← Σ_{i=1}^{n} f · [[E]]s_1..s_i.[[E^{*f}]]s_{i+1}..s_n
        Σ_{i=1}^{k} [[E]]_prefix s ← Σ_{i=1}^{n} f · [[E]]s_1..s_i · [[E^{*f}]]_prefix s_{i+1}..s_n
        Σ_{i=1}^{k} [[E]]_suffix s ← Σ_{i=1}^{n} f · [[E]]_suffix s_1..s_i · 1/(1-f) · [[E^{*f}]]s_{i+1}..s_n
        Σ_{i=1}^{k} [[E]]_infix s ← Σ_{i=1}^{n} f · [[E]]_suffix s_1..s_i · [[E^{*f}]]_prefix s_{i+1}..s_n
    else
        incorrect model
    return [[E_root]]_infix s
```

Algorithm 1: Unbounded reachability algorithm

4.2.2.2 *Bounded Probabilistic Reachability*

The bounded reachability establishes a limit to reach the desired point in the analysis. Answering the question, for example, "what is the probability that a system reaches

"*purchase complete*" action successfully in less than 5 steps?" requires the calculation of maximum length up to 5 when the execution meets success. In this case, one needs to consider all the length possibilities from 0 to 5 for every path. It can be denoted using bounded *Until* ($\mathcal{U}^{\leqslant n}$ with bound $n \in \mathbb{N}$). Reasoning on some formula, such as "$a\,\mathcal{U}^{\leqslant n}\,b$", searches the *prefixes* "ab", "aab" up to some bound. Therefore, we can use the *flat until* formula provided in Algorithm 1 by adding a loop of n to the formula.

The bounded formula in the form of "**true** $\mathcal{U}^{\leqslant n}\,a$" means that reaching an action "a" in maximum n steps accepts the *prefix* patterns in the sequences like "a", "$\neg aa$", "$\neg a\neg aa$", "$\neg a\neg a\neg aa$", ..., "$\underbrace{\neg a \ldots \neg a}_{n}\,a$" up to some bound n. Such a method requires the definition of *negation* in our framework. The probability that an action sequence s does not match E is

$$[\![E]\!]_{\text{not}}s = 1 - [\![E]\!]s$$

Similarly,

$$[\![E]\!]_{\text{not-prefix}}s = 1 - [\![E]\!]_{\text{prefix}}s.$$
$$[\![E]\!]_{\text{not-suffix}}s = 1 - [\![E]\!]_{\text{suffix}}s.$$
$$[\![E]\!]_{\text{not-infix}}s = 1 - [\![E]\!]_{\text{prefix}}s.$$

Recalling one of the advantages of SREs for the computation is local calculations for every SRE node. We also define a specific probability function $\mathcal{P}_{\text{length}}(E, n)$ where n is a natural number and E is an SRE.

Every term has a probability function for a certain length n for an execution (see Formula (49)). This function enables us to model check *bounded* $\mathcal{U}$ on an expression together with inclusion. With this function, we know the probability of being in the length of n in every SRE term.

Such a function gives the probability of reachable actions up to some length. More specifically, we search s in E and if it is included; in case of inclusion, we get the probability up to that length.

$$[\![E]\!]_{\text{length(n)}} = \begin{cases} 1, & \text{if } E = a \wedge n = 1 \\ 0, & \text{if } E = a \wedge n \neq 1 \\ \sum_k \left(\frac{n_k}{\sum_k n_k}\right) \cdot [\![E_k]\!]_{\text{length(n)}}, & \text{if } E = \sum_k E_k[n_k] \\ \sum_{i=0}^{n} [\![E_1]\!]_{\text{length}(i)} \cdot [\![E_2]\!]_{\text{length}(n-i)}, & \text{if } E = E1 : E2 \\ \sum_{i=1}^{n} [\![E]\!]_{\text{length}(i)} \cdot f^{(\frac{n}{i})} \cdot (1-f), & \text{if } E = E^{*f} \wedge n \neq 0 \\ 1 - f, & \text{if } E = E^{*f} \wedge n = 0 \end{cases} \tag{49}$$

In this case, n is the requested *length* and $p \in [0, 1]$ is the probability of E being n *length*. The detail description of function (49) is as follows.

- **Atomic actions:** Any action has the *length* of 1, hence the probability of an action a has the *length* 1 is 1.0 and the rest is 0.

$$\begin{aligned} [\![a]\!]_{\text{length(0)}} &= 0 \\ [\![a]\!]_{\text{length(1)}} &= 1 \\ &\vdots \\ [\![a]\!]_{\text{length(n)}} &= 0 \end{aligned} \tag{50}$$

- **Choice:** Every term has the possibility to occur; hence, the probability of E being *length* n is the probability of the chosen term.

$$\left[\!\left[\sum_i E_{i\,[n_i]}\right]\!\right]_{\text{length}(n)} = \sum_k \left(\frac{n_k}{\sum_k n_k}\right) \cdot [\![E_k]\!]_{\text{length}(n)} \tag{51}$$

- **Concatenation:** Every concatenation of two terms $E_1 : E_2$ increases the *length* by every combination of E_1 and E_2, and the *length* values are for every index up to n.

$$[\![E_1 : E_2]\!]_{\text{length}(n)} = \sum_{i=0}^{n} \left([\![E_1]\!]_{\text{length}(i)} \cdot [\![E_2]\!]_{\text{length}(n-i)} \right) \tag{52}$$

For instance the probability of $a : b$ being in the *length* of 2 is calculated as follows:

$$\begin{aligned}
[\![a : b]\!]_{\text{length}(2)} &= [\![a]\!]_{\text{length}(0)} \cdot [\![b]\!]_{\text{length}(2)} \\
&+ [\![a]\!]_{\text{length}(1)} \cdot [\![b]\!]_{\text{length}(1)} \\
&+ [\![a]\!]_{\text{length}(2)} \cdot [\![b]\!]_{\text{length}(0)}
\end{aligned}$$

- **Kleene closure:** Kleene term E^{*f} can have the same size of E for a specific *length* n with every combination of each iteration as a summation.

$$[\![E^{*f}]\!]_{\text{length}(n)} = \sum_{i=1}^{n} [\![E]\!]_{\text{length}(i)} \cdot f^{\left(\frac{n}{i}\right)} \cdot (1-f)$$

The termination probability without executing the system is defined as $1-f$ so that the *length* of the term E to become zero is also $1-f$.

As an example, let $E = \underbrace{(a : b)}_{E_x}{}^{*0.4}$ be an SRE on the alphabet $\Sigma = \{a, b\}$ and $E_x = a : b$. Then,

$$\begin{aligned}
[\![E]\!]_{\text{length}(0)} &= 1 - 0.4 = 0.6 && (53)\\
[\![E]\!]_{\text{length}(1)} &= \underbrace{[\![E_x]\!]_{\text{length}(1)}}_{0} \cdot 0.4^1 \cdot 0.6 = 0 \\
[\![E]\!]_{\text{length}(2)} &= \underbrace{[\![E_x]\!]_{\text{length}(1)}}_{0} \cdot 0.4^{\frac{2}{1}} \cdot 0.6 \\
&+ \underbrace{[\![E_x]\!]_{\text{length}(2)}}_{1} \cdot 0.4^{\frac{2}{2}} \cdot 0.6 = 0.24 && (54)\\
[\![E]\!]_{\text{length}(3)} &= 0 \\
[\![E]\!]_{\text{length}(4)} &= 0.096 \\
&\vdots
\end{aligned}$$

For *plus closure*, the probability of being *length* of zero is zero, and the rest remains similar.

4.2.3 *Generalized Algorithm*

Stochastic regular expressions are actually based on the probabilistic algebra defined in the study of Kumar et al. [54] and their language has been already proved that it is closed under the regular operations like concatenation, closure and choice. It is a complete partial order in which the operators are continuous. Recursive equations can inherently be solved in this algebra. As a result, to apply the model checking of SREs over PACTL, we defined recursive functions that execute the semantics of an SRE. These functions allow us to measure the probability of a string to be a *word*, *prefix*, *suffix*, or *infix* of an SRE term (see Table 3).

Words(E)	$\{a \mid a \in L(E)\}$
Prefix(E)	$\{a \mid a \in \Sigma^*$ and aw, where $aw \in L(E)\}$
Suffix(E)	$\{a \mid a \in \Sigma^*$ and wa, where $wa \in L(E)\}$
Infix(E)	$\{a \mid a \in \Sigma^*$ and waw', where $waw' \in L(E)\}$
Length(w)-$\lvert w \rvert$	$w = w_0 w_1 ... w_n$ is the number of included characters $\lvert w \rvert = n$
Where aw, wa, awa are the words concatenated by $a : w$, $w : a$ and $w : a : w'$ respectively with $a, w \in \Sigma$	

Table 3: Summary of definitions

As we mentioned above, a system is specified as an SRE tree that includes a root node and a finite set of nodes, which is formally defined as $T_E = (E_{root}, \mathcal{E}, \Sigma)$, where E_{root} is the root node, $\mathcal{E}$ is the finite set of all nodes, and Σ is the alphabet. Hence, verifying the root node E_{root} will result in the verifying the system (Algorithm 2).

Based on the semantics (27), every node up to the root E_{root} is equipped with the probability functions for the string calculations for the translated PACTL formula in Regex form (see the translation function in (26)).

4.3 ILLUSTRATIVE EXAMPLE

We provide an example system defined on a SRE specification (provided in Figure 15) that constitutes a root E_{root}, a set of SRE terms $\mathcal{E}$, and an alphabet Σ. Let us assume that we have a system that is composed of some web services aiming to achieve a message protocol. The subcomponents Service 1 (S_1) and Service 2 (S_2) are executing the login to the system and message sending, respectively.

Let a PACTL formula be $\mathcal{P}_{[0,0.04]}\Big((\mathbf{true})\,\mathcal{U}\,(\mathcal{X}_{\text{msgFail}})(\mathbf{true})\Big)$ as a system requirement for the analysis on T_E. The formula indicates the reachability analysis of the action "msgFail" on the root node $E_{root} = S$. The words reaching the "msgFail' from S are then recursively calculated over the words:

$$\begin{aligned}&\mathtt{Words}(S)_{[\text{reaching"msgFail"}]} \subset \mathtt{Words}(S) = \{\mathtt{start.login.sendMsg.msgFail}\ (0.0325),\\ &\mathtt{start.login.sendMsg.succes.sendMSg.msgFail}\ (0.008125),\\ &\mathtt{start.login.sendMsg.succes.sendMsg.succes.sendMsg.msgFail}\ (0.00203125), \ldots\}.\end{aligned}$$

The union is then

$$0.0325 \times \prod_{i=0}^{\infty} (0.25)^i = 0.0325 \in [0, 0.04]$$

$$S \models \mathcal{P}_{[0,0.04]}\Big((\mathbf{true})\,\mathcal{U}\,(\mathcal{X}_{\text{msgFail}})(\mathbf{true})\Big) \tag{56}$$

```
ModelCheck(E, propertySRE)
Data: An SRE tree T_E, propertySRE
Result: Model checking result
foreach E ∈ ℰ until reaching E_root do
    if E is an "action node" α then
        for ∀r_i ∈ propertySRE do
            hashInsert(true,1.0) ;
            if "α"= r_i then
                hashInsert(r_i,1.0);
            else
                hashInsert(r_i,0.0);
    else if E is a "choice node", E = E_1[n_1] + E_2[n_2] then
        for ∀r_i ∈ propertySRE do
            hashInsert(true,1.0) ;
            hashInsert(r_i, ∑ n_i/sum) ;
    else if E is a "concat node", E = E_1 : E_2 then
        for ∀r_i ∈ propertySRE do
            for ∀(w_0, p_0) ∈ Words(N_0) do
                for ∀(w_1, p_1) ∈ Words(N_1) do
                    result = check (w_o : w_1, p_1.p_2) ⊨ r_i ;
                    hashInsert(r_i,result);
    else if E is a "Kleene node", E = E^{*f} then
        WordsN = WordsN_0*WordsN_0;
        for ∀r_i ∈ propertySRE do
            result = check ∀w ∈ WordsN_0 ⊨ r_i until reaching a fix point
             starting from ∅ ;   /* x*y is the least fix-point solution of
             the equation: X = L_y ∪ (L_x : X) */
            hashInsert(r_i,result);
    else
        not defined
    return result;
```

Algorithm 2: Model checking SREs over the PACTL formula

The probabilistic functions are determined as provided in the reachability algorithm (Algorithm 1) and synthesized in a bottom up way by substitution of the SRE terms as follows. Such a technique allows reaching every calculation on each node locally. The idea is to calculate all information on every SRE term and to compose the solutions based on the operations.

Reasoning the probability of reaching action "msgFail" implies the calculation of the *infix* probability function (Algorithm 1). The *infix* function depends on the *prefix* and *suffix* functions. We have previously provided the execution of the *prefix* function in function (28); therefore, the execution of the *suffix* and *infix* functions are demonstrated in the following.

We start to substitute the results of actions, and we know that the only action that matches and includes "msgFail" is itself with probability 1; the probability of all other

$$
\begin{aligned}
&T_E = (E_{root}, \mathcal{E}, \Sigma)\\
&E_{root} = S\\
&\mathcal{E} = \{S, S_1, S_2, E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\\
&\Sigma = \{\mathtt{start}, \mathtt{login}, \mathtt{authenticationFail}, \mathtt{logout}, \mathtt{sendMsg},\\
&\mathtt{msgFail}, \mathtt{terminate}, \mathtt{success}\}\\
&S = S_1 : S_2\\
&S_1 = \mathtt{start} : \mathtt{login}\\
&S_2 = E_{1\,[65]} + E_{5\,[20]} + \mathtt{authenticationFail}_{[15]}\\
&E_1 = \mathtt{sendMsg} : E_2\\
&E_2 = E_{3\,[95]} + \mathtt{msgFail}_{[5]}\\
&E_3 = E_4 : E_7\\
&E_4 = E_6^{*0.25}\\
&E_5 = \mathtt{logout} : \mathtt{terminate}\\
&E_6 = \mathtt{success} : \mathtt{sendMsg}\\
&E_7 = \mathtt{success} : E_5
\end{aligned}
\tag{55}
$$

Figure 15: An example of an SRE model defining a simple message protocol

actions except, "msgFail" to match "msgFail" or have "msgFail" as *prefix*, *suffix*, and *infix* are 0.

$$
\begin{aligned}
&[\![\mathtt{msgFail}]\!]\mathtt{msgFail} = 1.0\\
&[\![\mathtt{msgFail}]\!]_{\text{infix,suffix,prefix}}\mathtt{msgFail} = 1.0
\end{aligned}
\tag{57}
$$

The probability functions of *matching*, *prefix*, *suffix*, and *infix* are calculated for every SRE term; and propagated until all terms are covered. We start by substituting the known probabilities of the actions "logout" and "terminate" in E_5 using the *infix probability function* (42).
($[\![\mathtt{logout}]\!]_{\text{infix,suffix,prefix}}\mathtt{msgFail} = 0.0$ and same for "terminate". Therefore, the union is also 0 and omitted in the text for the readability).

$$
\begin{aligned}
[\![E_5]\!]_{\text{infix}}\mathtt{msgFail} = {} & \overbrace{[\![\mathtt{logout}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![\mathtt{terminate}]\!]_{\text{prefix}}\varepsilon}^{1}\\
& + \overbrace{[\![\mathtt{logout}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} + \overbrace{[\![\mathtt{terminate}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0
\end{aligned}
\tag{58}
$$

The calculation of the *suffix*, the *prefix*, and the *matching* functions for E_5 are similar to (58), and ($[\![E_5]\!]_{\text{suffix,prefix}}\mathtt{msgFail} = 0$).

The *infix* function of E_7 is calculated by using the *prefix* function of E_5 and *suffix* function of "success" as follows. (The calculation of the *suffix*, the *prefix* and the *match-*

ing functions for E_7 are calculated similarly using E_5 and "success" and therefore $[\![E_7]\!]_{\text{suffix,prefix}}\mathtt{msgFail} = 0$).

$$[\![E_7]\!]_{\text{infix}}\mathtt{msgFail} = \overbrace{[\![\mathtt{success}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_5]\!]_{\text{prefix}}\varepsilon}^{1} + \overbrace{[\![\mathtt{success}]\!]_{\text{infix}}\varepsilon}^{0} + \overbrace{[\![E_5]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0 \tag{59}$$

The calculation of the *infix* function for E_6 is calculated using *suffix* function of "success" and *prefix* function of "sendMsg".

$$[\![E_6]\!]_{\text{infix}}\mathtt{msgFail} = \overbrace{[\![\mathtt{success}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{prefix}}\varepsilon}^{1} + \overbrace{[\![\mathtt{success}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} + \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0 \tag{60}$$

Using the Kleene function (48):

$$[\![E_4]\!]_{\text{infix}}\mathtt{msgFail} = 0.25 \cdot \overbrace{[\![E_6]\!]_{\text{infix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_6]\!]_{\text{prefix}}\varepsilon}^{1} + 0.25 \cdot \overbrace{[\![E_6]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0 \tag{61}$$

$$[\![E_3]\!]_{\text{infix}}\mathtt{msgFail} = \overbrace{[\![E_4]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_7]\!]_{\text{prefix}}\varepsilon}^{1} + \overbrace{[\![E_4]\!]_{\text{infix}}\mathtt{msgFail}}^{0} + \overbrace{[\![E_7]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0 \tag{62}$$

The *Infix*, *prefix*, and *suffix* probability functions are calculated for the choice SRE term (E_2) as follows:

$$[\![E_2]\!]_{\text{prefix}}\mathtt{msgFail} = \overbrace{[\![E_3]\!]_{\text{prefix}}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} + \overbrace{[\![\mathtt{msgFail}]\!]_{\text{prefix}}\mathtt{msgFail}}^{1} \cdot \frac{5}{100} = 0.05 \tag{63}$$

$$[\![E_2]\!]_{\text{suffix}}\mathtt{msgFail} = \overbrace{[\![E_3]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} + \overbrace{[\![\mathtt{msgFail}]\!]_{\text{suffix}}\mathtt{msgFail}}^{1} \cdot \frac{5}{100} = 0.05 \tag{64}$$

$$[\![E_2]\!]_{\text{infix}}\mathtt{msgFail} = \overbrace{[\![E_3]\!]_{\text{infix}}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} + \overbrace{[\![\mathtt{msgFail}]\!]_{\text{infix}}\mathtt{msgFail}}^{1} \cdot \frac{5}{100} = 0.05 \tag{65}$$

Using the calculations of E_2, we obtain the results of E_1 as follows:

$$\begin{aligned}[\![E_1]\!]_{\text{suffix}}\mathtt{msgFail} &= \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]\varepsilon}^{0} \\ &+ \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{suffix}}\varepsilon} \cdot \overbrace{[\![E_2]\!]\mathtt{msgFail}}^{0} + \overbrace{[\![E_2]\!]_{\text{suffix}}\mathtt{msgFail}}^{0.05} \\ &- 0 \cdot 0.05 = 0.05\end{aligned} \tag{66}$$

$$\begin{aligned}[\![E_1]\!]_{\text{prefix}}\mathtt{msgFail} &= \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{prefix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]\varepsilon}^{1} \\ &+ \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{prefix}}\varepsilon}^{1} \cdot \overbrace{[\![E_2]\!]\mathtt{msgFail}}^{0} = 0\end{aligned} \tag{67}$$

In the calculation of the *infix* for the concatenation operation, the possibilities of the events, which are the decompositions of E_1 can include "msgFail" in different positions, are also taken into account.

$$\begin{aligned}[\![E_1]\!]_{\text{infix}}\mathtt{msgFail} &= \underbrace{\overbrace{[\![\mathtt{sendMsg}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]_{\text{prefix}}\varepsilon}^{1}}_{0} \\ &+ \overbrace{[\![\mathtt{sendMsg}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} + \overbrace{[\![E_2]\!]_{\text{infix}}\mathtt{msgFail}}^{0.05} = 0.05 \\ &- 0 \cdot 0 - 0 \cdot 0.05 - 0 \cdot 0.05 + 0 \cdot 0 \cdot 0.05 = 0.05\end{aligned} \tag{68}$$

The all choice terms are calculated respectively.

$$\begin{aligned}[\![S_2]\!]_{\text{infix}}\mathtt{msgFail} &= \overbrace{[\![E_1]\!]_{\text{infix}}\mathtt{msgFail}}^{0.05} \cdot \frac{65}{100} \\ &+ \overbrace{[\![E_5]\!]_{\text{infix}}\mathtt{msgFail}}^{0} \cdot \frac{20}{100} \\ &+ \overbrace{[\![\mathtt{authenticationFail}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} \cdot \frac{15}{100} = 0.05 \cdot 0.65 = 0.035\end{aligned} \tag{69}$$

The results for S_1 can be easily obtained through "start" and "login" actions.

$$\begin{aligned}[\![S_1]\!]_{\text{infix}}\mathtt{msgFail} &= \overbrace{[\![\mathtt{start}]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![\mathtt{login}]\!]_{\text{prefix}}\varepsilon}^{1} \\ &+ \overbrace{[\![\mathtt{start}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} + \overbrace{[\![\mathtt{login}]\!]_{\text{infix}}\mathtt{msgFail}}^{0} = 0\end{aligned} \tag{70}$$

The reachability of "msgFail" is then:

$$\begin{aligned}[\![S]\!]_{\text{infix}}\mathtt{msgFail} &= \underbrace{\overbrace{[\![S_1]\!]_{\text{suffix}}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![S_2]\!]_{\text{prefix}}\varepsilon}^{1}}_{A} \\ &+ \underbrace{\overbrace{[\![S_1]\!]_{\text{infix}}\mathtt{msgFail}}^{0}}_{B} + \underbrace{\overbrace{[\![S_2]\!]_{\text{prefix}}\mathtt{msgFail}}^{0.035}}_{C} = 0.035 \\ &- A \cdot B - B \cdot C - A \cdot C + A \cdot B \cdot C = 0.035\end{aligned} \tag{71}$$

The same system can be modelled as a PRA, which is equivalent to the SRE specification (The equivalence relationship and an equivalent PRA to the SRE model are provided in the following Chapter 5). We used a widely used probabilistic model checker PRISM [100] to validate the outcome of the model checking result. For this reason, we model this PRA in the Prism editor as a DTMC (the action labels are moved to states [8]) and model check the reachability of "msgFail". The result of the PRISM engine is 0.0325 as same as our SRE checker.

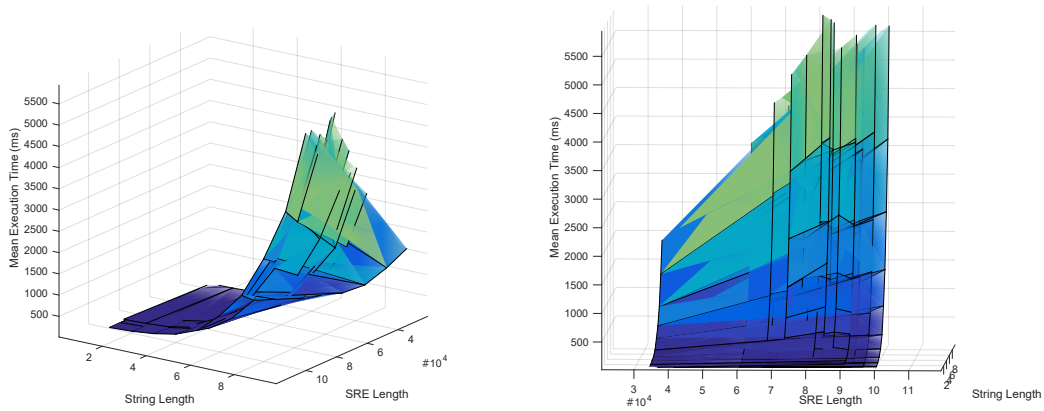
4.4 PRELIMINARY EVALUATION

We generate some random SRE models and search strings with different sizes to evaluate the SRE model checking framework. The generation of SRE models is achieved via a stochastic context free grammar, which is actually a specific form of weighed grammars [44]. By this grammar, a probability distribution for each production rule is attached, and probabilities are normalized at the level of each non-terminal (formula (72)) using the SRE syntax (equation (18)). Hence we could generate meaningful random SREs over the created grammar.

$$\begin{aligned} S &\rightarrow E(1.0) \\ E &\rightarrow \sum_i E_i(n_i)(0.4) \\ E &\rightarrow E_1 : E_2(0.4) \\ E &\rightarrow E^{*f}(0.1) \\ E &\rightarrow \alpha(0.1) \end{aligned} \tag{72}$$

We adapt and configure this distribution, e.g., decreasing the probability of action can create longer SREs. The data demonstrated below in the figures are created from a data set where SRE sizes (length) range from 1,000 to 700,000.

For each generated SRE, we apply the reachability analysis ($\Diamond s$) using the *infix function (function (42))* for a range of string (s) size from 1 up to 10 size. One can observe in Figure 16b (in 3D-view) that the execution time (in milliseconds) is mostly affected by the search string size, in other words, logic property elements. The same data set (in 2D-view) can be seen in Figure 16a to demonstrate the high execution time performance even for the larger SREs. The execution time and the memory eventually increase with the search string size.



(a) Effect of SRE length and the size of the search string on the execution time for randomly generated models.

(b) Increasing effect of the string size on the execution time from a different view for the same data.

Figure 16: Random SREs up to 700,000 length for the size of the string sought from 1 up to 10.

4.5 DISCUSSION

In this section, we have introduced a formalism for model checking probabilistic properties with stochastic regular expressions. The proposed approach in this paper is similar to Kleene like languages [19], and tree automata [34]; however we use it in the model checking context by questioning if the disjoint sets of words satisfy the formula with a probabilistic branching logic.

Let us summarize the process of the model checking PACTL on SREs. We describe a translation of PACTL word formulas into stochastic regular expressions and formalize the semantics as a satisfaction relation between the words of the SRE operations. The essential idea is lying on this semantics in section 4.2.1, where we find the meaning of $w \models$ `propertySRE`, such that w is a word of a SRE. When constructing the SRE tree during parsing, we calculate the word sets and the fix points of these sets to check if they satisfy `propertySRE`. We propagate the probabilities of satisfying words with hash functions to the parent node in the tree. Hence, `propertySRE` is recursively checked on a parse tree. The probabilities are summed up whenever the SRE choice node is reached and the algorithm terminates when the tree is traversed bottom up. The quantitative verification Framework is developed on the top of a parser using the SRE grammar attached with some verification values. Once an initial model is parsed, the SRE tree is equipped with the verification results that are maps of the probabilistic values. In the upcoming chapter (Chapter 6), we will use the benefits of modularity of such formalism for the incremental verification.

5 EQUIVALENCE BETWEEN PROBABILISTIC RABIN AUTOMATA AND STOCHASTIC REGULAR EXPRESSIONS

In this section, the equivalence relations between a probabilistic Rabin automation (PRA) and a stochastic regular expression (SRE) are provided (Figure 17). Such equivalence relations are achieved via transformations similar to the conversion algorithms between finite state automata and regular expressions. In the following approach, the probabilities are additionally included in determining the equivalence relations.

To validate the proposed approach, we first present theorems and proofs regarding that an SRE/a PRA can be transformed into a PRA/an SRE defining the translation semantics of the SRE. Secondly, we establish another theorem that consolidates such a transformed model (target model) produces equal measurements with the probability function of the source model. In other words, an SRE and its corresponding PRA produce the same measurements for the same execution paths.

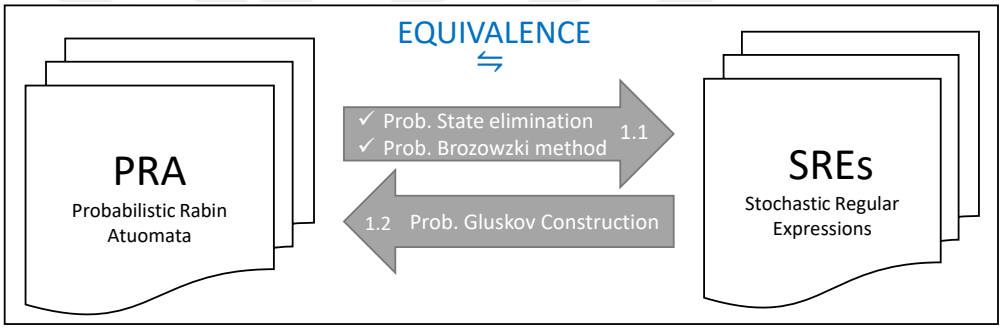


Figure 17: PRA-SRE transformations overview

5.1 FROM A PRA TO AN SRE

The transformation from a finite automata (FA) to a regular expression (regex) is a long-standing research problem with respect to the time and space complexities as well as the regex size. Algorithms, such as state elimination[40] and Brzozowski's Algebraic Method[21], are commonly used techniques. Both techniques are explained in detail below.

5.1.1 *Conversion Using State Elimination*

Let Σ be a finite set of symbols, called alphabet, and Σ^* the words over Σ. A regular expression α over Σ represents a regular language $L(\alpha) \subset \Sigma^*$ for which, according to Kleene's theorem [96], there exists a finite automaton (FA) accepting this language. An FA is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. We say that an FA is *uniform* [32] if it has a unique initial state with no

incoming transitions and a unique final state with no outgoing transitions. Uniforming an FA can be performed as follows:

- If $q_0 \in F$ or if there exists an $a \in \Sigma$ and $q \in Q$ so that $\delta(q, a) = q_0$, then add a new state i to Q with $\delta(i, \epsilon) = q_0$ and set i as the new initial state instead of q_0.
- If $|F| \geqslant 1$ or if there exists $p \in F$, $a \in \Sigma$ and $q \in Q$ so that $\delta(p, a) = q$, then add a new state f to Q, add the transition $\delta(p, \epsilon) = f$ for all $p \in F$ and set $F = \{f\}$.

The state elimination algorithm [40] takes a uniformed FA as an input and produces a regular expression (describing its accepted language) as an output.

Let $\mathcal{R}(\Sigma)$ be the universe of regular expressions over a given alphabet Σ, then the transition function is generalized to $\delta : Q \times \mathcal{R}(\Sigma) \to Q$. In other words, regular expressions used as transition labels are utilized to keep track of eliminated states.

The final output of the state elimination method has two states. The label on the transition between those states represents the output regular expression.

In algorithm 3, let α_{xy} be the regular expression for the transition from state x to state y, and $A = (Q, \Sigma, \delta, q_0, F)$ be an FA, we refer to its uniformed form as $A' = (Q', \Sigma, \delta', i, \{f\})$ during state elimination.

Input: A uniform FSA A'.
Output: A regular expression defining the regular language accepted by A'.

1: **repeat**
2: Randomly choose a state $k \in Q' \setminus \{i, f\}$ and remove it from A'.
3: Then, for all pairs $p, q \in Q' \setminus \{k\}$ the new regular expression α'_{pq} for the transition from p to q is:

$$\alpha'_{pq} = \alpha_{pq} + \alpha_{pk}\alpha^*_{kk}\alpha_{kq}$$

4: **until** There are only the initial state i and the final state f left.
5: **return** α_{if}

Algorithm 3: State elimination algorithm [62]

The case of transforming an FA into an equivalent regex can be extended to converting a PRA into an SRE.

State elimination for PRA follows the same basic principles as for FSAs. The extended steps for the probabilistic version of the state elimination appear with the probability calculation. In the beginning, the probabilities of all outgoing transitions are recalculated where the probability of the loops remains unchanged. During the "label join", the probabilities are multiplied and uniformed to the rates (e.g., constant=1000) when expressions are operated with concatenation, star, and choice, respectively. Figure 18 demonstrates the probabilistic extension of state elimination by means of an example. A model-based implementation of the algorithm is published in [62].

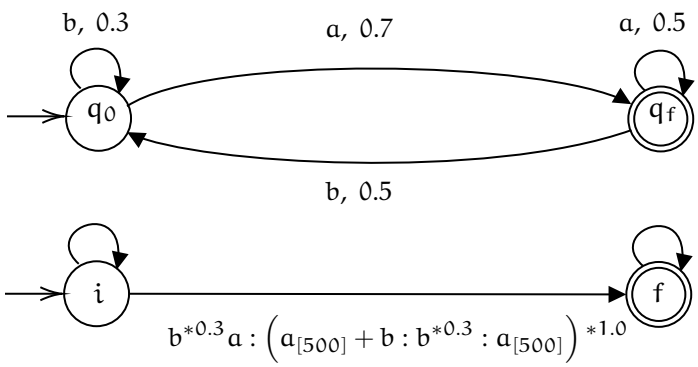


Figure 18: An example of state elimination with probabilities

5.1.2 *Conversion Using Brzozowski's Algebraic Method: Regular Expression Equation Systems Solving*

We recall the Brzozowski's method [21] by providing the algorithm that transforms an automata to a regex term. For a given FA $(Q, \Sigma, \delta, q_0, F)$, the algorithm is proceed as follows:

- Create a system of regular equations as one equation for each state q_i and denote the corresponding the variable as R_i. For each state q_i in automata M, the equation for regex R_i is a union of terms obtained from the outgoing states and the actions. The term is aR_j is constructed for each term with a transition a from q_i to q_j. If R_i is a final state, ϵ is assigned as a term, which leads to a system of equations in the form

$$\begin{aligned} R_1 &= a_1^1 R_1 + a_2^1 R_2 + \dots \\ R_2 &= a_1^2 R_1 + a_2^2 R_2 + \dots \\ R_3 &= a_1^3 R1 + a_2^3 R_2 + \dots + \epsilon \\ \dots & \\ R_n &= a_1^n R_1 + a_2^n R_2 + \dots + \epsilon \end{aligned} \tag{73}$$

- Solve the system using Arden's lemma [4].

> **Arden's Lemma**
>
> Given an equation of the form $X = AX + B$ where $\epsilon \notin A$, the equation has the solution $X = A^*B$.

- The regex equivalent to the initial state q_0 will be regex (R_1) representing the automata.

We need a probabilistic extension of Brzozowski's algebraic method and *Arden's lemma* to achieve the transformation from a PRA to an SRE using the equations of SREs via *Arden's lemma*. Thus, the following paragraph presents the required method.

5.1.3 *Probabilistic Brzozowski's Algebraic Method*

Let $PRA = (\Sigma, Q, q_0, M, F)$ be a PRA where Σ is the alphabet, Q is the finite states set, q_0 is the starting state, $M : Q \times \Sigma \to [0,1]^{n+1}$① is a transition function that maps state-action tuple (q, a) into a probabilistic value set (the transition probabilities table is implicitly determined) such that $(q, a) \in Q \times \Sigma$, and M is the set that includes all probabilistic values (p) from a state q with every action:

$$M(q,a) = \{p(q,a_1), \cdots, p(q,a_n)\},$$
$$0 \leqslant p(q,a_i), \qquad \sum_i p(q,a_i) = 1$$

All outgoing transition probabilities from a state sum up to 1.0, and F is the set of final states.

As described in [21], we initially build an equation system of a given PA for every state including probabilities. The variables $x_0, x_1, \ldots, x_n$ represent the states; and $a_0, a_1, \ldots, a_n$ are transition labels or actions, and p_{i_j} corresponds to the probabilities of a given PRA. Accordingly, every *action-target state-probability* triple for each state (source) is constructed (if exists) in the linear equation system as follows. Hence, all outgoing transitions are encoded with choice operator (+) including probabilities.

$$\begin{aligned} x_0 &= a_0 x_0(p_{0_0}) + a_1 x_1(p_{0_1}) + \ldots \\ x_1 &= b_0 x_0(p_{1_0}) + b_1 x_1(p_{1_1}) + \ldots \\ &\ldots \\ &\text{if } x_i \in F \text{ then } x_i = \varepsilon \end{aligned}$$

Each equation x_i *is an SRE for the state* q_i. Once the equation system is solved by the *substitution* method, x_0 is representing the whole expression. The probabilities cannot be utilized directly because, by the definition of SRE, every choice term has an integer choice rate n_i and the probability of a transition between states q_i and q_j, $P(q_i, q_j, a)$ (where $a \in \Sigma$ is the transition label) is calculated by getting the ratio between $n_{i,j}$ and $\sum_j n_{i,j}$ for that choice operation:

$$P(q_i, q_j, a) = \frac{n_{i,j}}{\sum_j n_{i,j}}$$

Because the probabilities are rational, we can assure that $\exists x, y \in \mathbb{N}^+$ such that $\frac{x}{y} = P(q_i, q_j, a)$. If we choose $n_{i,j} = P(q_i, q_j, a)10^k$ (the value 10^k is multiplied all choice elements of the choice operation), there exists a number k such that $n_{i,j}$ becomes a natural number.

The equations are removed step by step. We start eliminating the equations for the final states by substituting them with ε and placing them into the other equations.

① $[0,1]^{n+1}$ is the set of all $n+1$ tuples $x_0, \ldots, x_n$ where $0 \leqslant x_i \leqslant 1$ [125].

Now each equation takes the following form (the sum notation Σ represents the choice operator (+) in a compact form for the following equation systems):

$$x_i = \left(\sum_{q_j \in Q \setminus F, a \in \Sigma} (a : x_j)_{[n_{i,j}]} \right) + \left(\sum_{q_j \in F, a \in \Sigma} a_{[n_{i,j}]} \right)$$

If we have the form of $x = a : x + b$, x is eliminated by *Arden's lemma* [4]. Accordingly, the equation $x_i = ax_i + b$ has the solution $x_i = a^* b$. In SRE version, namely $x_i = ax_i(r) + b(p)$, has the solution $x_i = a^{*\frac{r}{r+p}} b$, where r and $p \in \mathbb{N}^+$. If x_m takes the form as follows and occurs on the right side, then x_m component represents the self loops in PRA.

$$x_m = \left(\sum_{\substack{q_j \in Q \setminus F \\ q_j \neq q_m}} (A : x_j)_{[n_{m,j}]} \right) + \left(\sum_{q_j \in F} B_{[n_{m,j}]} \right) + \sum_{a \in \Sigma} (C : x_m)_{[n_{m,m}]}$$

where A, B, C are string sequences $s_1 s_2 \cdots s_n$ such that $s_i \in \Sigma$

We separate the recurring element (*Loop*), and the rest of choice elements remain the same (*Rest*).

$$\overbrace{\sum (A : x_m)_{[n_{m,m}]}}^{\text{Loop}} + \overbrace{\left(\sum_{\substack{q_j \in Q \setminus F \\ q_j \neq q_m}} (B : x_j)_{[n_{m,j}]} \right) + \left(\sum_{q_j \in F} C_{[n_{m,j}]} \right)}^{\text{Rest}}$$

During this separation, the rules such as, the distributivity, associativity and commutativity might be applied using the equivalence laws of Kleene's algebra (Chapter 2). These rules need to be adapted to the SRE syntax. Thus, the following set of rules hold because of the probabilistic nature of SREs; where e, f, and g are some SREs and r, r_1, r_2, p and q are the integer values that represent the rates of SREs.

Equivalence relations for SREs

$$e_{[r_1]} + (f_{[p]} + g_{[q]})_{[r_2]} = e_{[\frac{r_1}{r_1+r_2} \cdot 10^k]} + f_{[\frac{p}{p+q} \cdot \frac{r_1}{r_1+r_2} \cdot 10^k]} + g_{[\frac{q}{p+q} \cdot \frac{r_1}{r_1+r_2} \cdot 10^k]} \tag{74}$$

e.g., $e_{[2]} + (f_{[3]} + g_{[7]})_{[8]} = e_{[0.2 \cdot 10^2]} + f_{[0.24 \cdot 10^2]} + g_{[0.56 \cdot 10^2]}$.

$$e_{[r]} + f_{[p]} = f_{[p]} + e_{[r]} \tag{75}$$

$$e_{[r_1]} + e_{[r_2]} = e_{[r_1+r_2]} \tag{76}$$

$$e : (f : g) = (e : f) : g \tag{77}$$

$$e : (f_{[p]} + g_{[q]}) = e : f_{[p]} + e : g_{[q]} \tag{78}$$

$$(e_{[r]} + f_{[p]}) : g = e : g_{[r]} + f : g_{[p]} \tag{79}$$

The elimination of the loop element x_m is then achieved by the Lemma 5.1 as follows:

$$\left(\sum (A : x_m)\right)^{*\frac{n_{m,m}}{n_{sum}}} : \left(\sum_{\substack{q_j \in Q \setminus F \\ q_j \neq q_m}} (B : x_j)_{[n_{m,j}]} + \sum_{q_j \in F} C_{[n_{m,j}]} \right)$$

The substitution continues until x_0 has no variables; hence, the equivalent SRE is obtained as an SRE of x_0 at the end.

The crucial point of this conversion is to prove the **probabilistic** setting of *Arden's lemma* is correct. The following lemma (Lemma 5.1) serves for this purpose. The probabilistic matching function of an SRE for a string $s \in \Sigma^*$, which is used in the following theorems, is provided in the definition 5.1 as a remainder from Chapter 2.

Definition 5.1 (Probability of a word (a reminder from Chapter 4))
The probabilistic matching function $[\![E]\!]s$ for every possible SRE term is calculated recursively, where $s = s_1, ..s_n \in \Sigma^*$ is a string and s_i is an arbitrary symbol.

- ***Atomic actions:***

$$\begin{aligned} [\![a]\!]s &= 1, \quad \textit{if} \quad a = s \\ [\![a]\!]s &= 0, \quad \textit{if} \quad a \neq s \end{aligned} \tag{80}$$

- ***Choice:***

$$\left[\!\!\left[\sum E_{i[n_i]} \right]\!\!\right] s = \sum_k \left(\frac{n_k}{\sum n_k} \right) \cdot [\![E_k]\!]s \tag{81}$$

- ***Concatenation:***

$$[\![E_1 : E_2]\!]s = \sum_{i=0}^{n} \left([\![E_1]\!]s_0 \dots s_i \cdot [\![E_2]\!]s_{i+1} \dots s_n \right)$$

- ***Kleene closure:***

$$\begin{aligned} [\![E^{*f}]\!]\varepsilon &= 1 - f \\ [\![E^{*f}]\!]s &= \sum_{i=1}^{n} \left(f \cdot [\![E]\!]s_1 \dots s_i \cdot [\![E^{*f}]\!]s_{i+1} \dots s_n \right) \\ &\quad + f \cdot [\![E]\!]s \cdot [\![E^{*f}]\!]\varepsilon \end{aligned} \tag{82}$$

- ***+Closure:***

$$\begin{aligned} [\![E^{+f}]\!]s &= \sum_{i=1}^{n} \left([\![E]\!]s_1 \dots s_i . [\![E^{*f}]\!] \times s_{i+1} \dots s_n \right) \\ &\quad [\![E]\!]s \cdot [\![E^{*f}]\!]s \end{aligned} \tag{83}$$

Lemma 5.1 (Probabilistic extension of Arden's Lemma)
Let L_1, L_2, L_3, *and* L_4 *be stochastic, regular languages over alphabet* Σ. *Then,*

$$\begin{aligned} L_1 =& (L_3 : L_1)_{[n_1]} + L_{4\,[n_2]} \text{ and } n_1, n_2 \in \mathbb{N} \\ L_2 =& L_3^{*(\frac{n_1}{n_1+n_2})} : L_4 \\ \Rightarrow& L_1 = L_2 \end{aligned}$$

holds.

Proof.— Proof by substitution to show the equivalence over the probability matching function $[\![E]\!]$:

Let $s \in \Sigma^*$ and $f = \frac{n_1}{n_1+n_2} \in [0,1] \subset \mathbb{R}$. Initially, $[\![L_2]\!]s$ is expanded as follows:

$$\begin{aligned} [\![L_2]\!]s &= [\![{L_3}^{*f} : L_4]\!]s(\text{ expanded as follows using the probability function 83:}) \\ &= \sum_{i=0}^{n} \left[\!\!\left[\epsilon_{[n_2]} + L_3 : L_{3\,[n_1]}^{*f}\right]\!\!\right] s_1 \dots s_i \cdot [\![L_4]\!]s_{i+1} \dots s_n \\ &= (1-f) \cdot \sum_{i=0}^{n} [\![\epsilon]\!]s_1 \dots s_i \cdot [\![L_4]\!]s_{i+1} \dots s_n + \\ & f \cdot \sum_{i=0}^{n} [\![L_3 : L_3^{*f}]\!]\, a_1 \dots s_i \cdot [\![L_4]\!]s_{i+1} \dots s_n \\ &= (1-f) \cdot [\![L_4]\!]s + f \cdot [\![L_3 : \overbrace{L_3^{*f} : L_4}^{=L_2}]\!]s \\ &= [\![L_4[n_2] + L_3 : L_{2\,[n_1]}]\!]\, s = [\![L_3 : L_{2\,[n_1]} + L_{4\,[n_2]}]\!]\, s \end{aligned}$$

hence, $L_1 = L_2$. □

The above-mentioned conversion shows that the construction of SREs from PRA is possible. The following theorem demonstrates that both models produce the same measurements for the same execution paths.

Theorem 5.1 (Equivalence of probabilistic matching function for a PRA and an SRE)
Let $PRA_i = (\Sigma, Q, q_i, P, F)$ *be a PRA whose initial state* q_i, *and* x_i *is the SRE constructed from the corresponding state* (q_i) *as described above. Then,*

$$[\![PRA_i]\!]s = [\![x_i]\!]s$$

holds, where $s = s_1 s_2 \dots s_n$ *is a string.*

Proof.— The probability that PRA_i matches a string is the sum of all probabilities calculated for all paths from q_i satisfying s. The probability of a string for a path σ from q_i satisfying s is obtained by multiplying all probabilities between every state until the path matches s.

Formally, $[\![PRA_i]\!]s = \sum_{\sigma \in PRA_i} \prod_{k=i}^{|\sigma|} P(q_k, q_{k+1}, a)$ is the sum of the probabilities of all paths through the PRA, which consume s. We show the equivalence by induction over the length of a path:

- For the trivial case: if the path length $n = 0$, the string is ε and every path ends in a final state, $q_i = q_f \in F$. Therefore, $[\![PRA_i]\!]\epsilon = 1 = [\![x_i]\!]\epsilon$.

- Let $\sigma = q_i a \sigma'$ be a path of length $n+1$ in $PRA_0 = PRA_1[n_1] + \cdots PRA_n[n_n]$ (representing the whole PRA), with $a \in \Sigma$ and σ' being a path of length n in PRA_0 from q_j to q_f with $q_f \in F$. The probabilistic matching of a PRA for a path is calculated by the sum of the probabilities of the possible paths. We show that $[\![PRA_i]\!]\sigma = [\![x_j]\!]\sigma$ as follows. By induction hypothesis, $[\![PRA_j]\!]\sigma' = [\![x_j]\!]\sigma'$ is given. Then,

$$
\begin{aligned}
[\![PRA_i]\!]\sigma &= \prod_{k=1}^{|\sigma|} P(q_k, q_{k+1}, a_k) \\
&= P(q_i, q_j, a) \cdot \prod_{k=1}^{|\sigma'|} P(q_k, q_{k+1}, a_k) \\
&= \frac{n_{i,j}}{\sum_{j'} n_{i,j'}} [\![PRA_j]\!]\sigma' \\
&= \frac{n_{i,j}}{\sum_{j'} n_{i,j'}} [\![PRA_j]\!]\sigma' = [\![x_i]\!]\sigma
\end{aligned}
$$

□

5.1.4 *Illustrative Example: From a PRA to an SRE*

We demonstrate a simple communication protocol modeled in PRA, where the probabilities are arbitrary (inspired by the parametric DTMC model from [48]) and apply the conversion algorithm step-by-step.

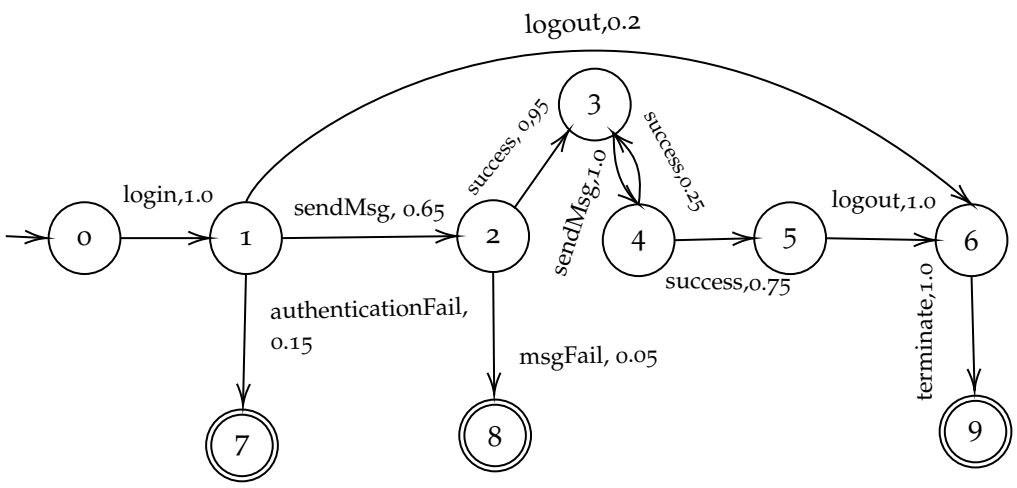


Figure 19: Example PRA representing a simple protocol

Let us constitute the equation system of PRA demonstrated in Figure 19 by denoting the states as variables x_i. Note that the probability 1.0 does not appear since it is represented as the *concatenation* operation in the equations.

$$
\begin{aligned}
x_0 &= \mathtt{login} : x_1 \\
x_1 &= \mathtt{sendMsg} : x_{2(0.65)} + \mathtt{logout} : x_{6(0.2)} + \mathtt{authenticationFail} : x_{7(0.15)} \\
x_2 &= \mathtt{success} : x_{3(0.95)} + \mathtt{msgFail} : x_{8(0.05)} \\
x_3 &= \mathtt{sendMsg} : x_4 \\
x_4 &= \mathtt{success} : x_{3(0.25)} + \mathtt{success} : x_{5(0.75)} \\
x_5 &= \mathtt{logout} : x_6 \\
x_6 &= \mathtt{terminate} : x_9 \\
x_7 &= \varepsilon \\
x_8 &= \varepsilon \\
x_9 &= \varepsilon
\end{aligned}
\tag{84}
$$

After the first iteration by applying the substitution method (elimination of ε) and converting probabilities to integers (the constant number is chosen 100 for every choice operator in this example), we obtain the following equation system:

$$
\begin{aligned}
x_0 &= \mathtt{login} : x_1 \\
x_1 &= \mathtt{sendMsg} : x_{2[65]} + \mathtt{logout} : x_{6[20])} + \mathtt{authenticationFail}_{[15]} \\
x_2 &= \mathtt{success} : x_{3[95]} + \mathtt{msgFail}_{[5]} \\
x_3 &= \mathtt{sendMsg} : x_4 \\
x_4 &= \mathtt{success} : x_{3[25]} + \mathtt{success} : x_{5[75]} \\
x_5 &= \mathtt{logout} : x_6 \\
x_6 &= \mathtt{terminate}
\end{aligned}
\tag{85}
$$

After the substitution of x_6, x_5, and x_4 (every substitution occurs with parentheses for a correct representation), we obtain the following equations, respectively:

$$
\begin{aligned}
x_0 &= \mathtt{login} : x_1 \\
x_1 &= \mathtt{sendMsg} : x_{2[65]} + (\mathtt{logout} : \mathtt{terminate})_{[20]} + \mathtt{authenticationFail}_{[15]} \\
x_2 &= \mathtt{success} : x_{3[95]} + \mathtt{msgFail}_{[5]} \\
x_3 &= \mathtt{sendMsg} : x_4 \\
x_4 &= \mathtt{success} : x_{3[25]} + \mathtt{success} : x_{5[75]} \\
x_5 &= \mathtt{logout} : \mathtt{terminate}
\end{aligned}
\tag{86}
$$

Substitution of x_5 in the equation x_4 is as follows:

$$
\begin{aligned}
x_0 &= \mathtt{login} : x_1 \\
x_1 &= \mathtt{sendMsg} : x_{2[65]} + (\mathtt{logout} : \mathtt{terminate})_{[20]} + \mathtt{authenticationFail}_{[15]} \\
x_2 &= \mathtt{success} : x_{3[95]} + \mathtt{msgFail}_{[5]} \\
x_3 &= \mathtt{sendMsg} : x_4 \\
x_4 &= \mathtt{success} : x_{3[25]} + (\mathtt{success} : \mathtt{logout} : \mathtt{terminate})_{[75]}
\end{aligned}
\tag{87}
$$

Substitution of x_4 in the equation x_3 is as follows:

$$\begin{aligned} x_0 &= \mathtt{login} : x_1 \\ x_1 &= \mathtt{sendMsg} : x_{2[65]} + \mathtt{logout} : \mathtt{terminate}_{[20]} + \mathtt{authenticationFail}_{[15]} \\ x_2 &= \mathtt{success} : x_{3[95]} + \mathtt{msgFail}_{[5]} \\ x_3 &= \mathtt{sendMsg} : \big(\mathtt{success} : x_{3[25]} + (\mathtt{success} : \mathtt{logout} : \mathtt{terminate})_{[75]}\big) \end{aligned} \tag{88}$$

Using probabilistic Arden's Lemma (5.1), the recursion in x_3 is eliminated.

$$\begin{aligned} x_0 &= \mathtt{login} : x_1 \\ x_1 &= \mathtt{sendMsg} : x_{2[65]} + \mathtt{logout} : \mathtt{terminate}_{[20]} + \mathtt{authenticationFail}_{[15]} \\ x_2 &= \mathtt{success} : x_{3[95]} + \mathtt{msgFail}_{[5]} \\ x_3 &= \underline{(\mathtt{sendMsg} : \mathtt{success})^{*0.25}} : (\mathtt{success} : \mathtt{logout} : \mathtt{terminate}) \end{aligned} \tag{89}$$

Now, we can substitute x_3 in x_2 as follows:

$$\begin{aligned} x_0 &= \mathtt{login} : x_1 \\ x_1 &= \mathtt{sendMsg} : x_{2[65]} + \mathtt{logout} : \mathtt{terminate}_{[20]} + \mathtt{authenticationFail}_{[15]} \\ x_2 &= \mathtt{success} : \big((\mathtt{sendMsg} : \mathtt{success})^{*0.25} : (\mathtt{success} : \mathtt{logout} : \mathtt{terminate})\big)_{[95]} \\ &+ \mathtt{msgFail}_{[5]} \end{aligned} \tag{90}$$

Similarly, x_2 is substituted in x_1.

$$\begin{aligned} x_0 &= \mathtt{login} : x_1 \\ x_1 &= \mathtt{sendMsg} : \big(\mathtt{success} : ((\mathtt{sendMsg} : \mathtt{success})^{*0.25} \\ &: (\mathtt{success} : \mathtt{logout} : \mathtt{terminate}))_{[95]} + \mathtt{msgFail}_{[5]}\big)_{[65]} \\ &+ \mathtt{logout} : \mathtt{terminate}_{[20]}) + \mathtt{authenticationFail}_{[15]} \end{aligned} \tag{91}$$

Finally, the SRE represented by x_0 is equivalent to the PRA given in Figure 19.

$$\boxed{\begin{aligned} x_0 = \mathtt{login} : \big(&\mathtt{sendMsg} : \big(\mathtt{success} : ((\mathtt{sendMsg} : \mathtt{success})^{*0.25} \\ &: (\mathtt{success} : \mathtt{logout} : \mathtt{terminate}))_{[95]} \\ &+ \mathtt{msgFail}_{[5]}\big)_{[65]} + \mathtt{logout} : \mathtt{terminate}_{[20]}) + \mathtt{authenticationFail}_{[15]}\big) \end{aligned}} \tag{92}$$

5.2 FROM AN SRE TO A PRA

There exist various algorithms that construct a finite state automaton from a given regular expression. For instance, *Thompson's construction* algorithm [119] constructs a non-deterministic finite automaton (NFA) with additional ε transitions [138] in complexity of $\mathcal{O}(|E|^2)$. On the other hand, Glushkov's construction algorithm constructs an NFA with ε-free transitions if ε is not included in the language. A well-summarized comparison between Thomson and Glushkov automata is presented in [65]. The theoretical background for this construction is provided in Chapter 2. Still, the definitions of the Glushkov's construction algorithm are summarized using an example.

		Successors	
→ The set 'first' (initial states)	$\emptyset$	b_1	c_4
← Final state	b_1	a_2	$\emptyset$
	a_2	b_3	
← Final state	b_3	$\emptyset$	a_2
← Final state	c_4	$\emptyset$	

(93)

Table 4: An example of Glushkov sets

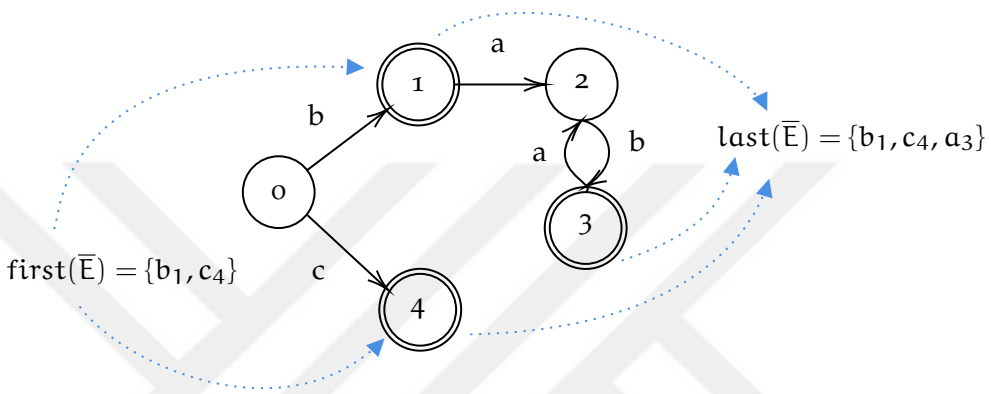


Figure 20: The FSM constructed from regex $E = b(ab)^* + c$

Example 5.1 (Reminder: Glushkov construction)
An example representing the definitions referring to the sets Pos, first, last, and follow are given below:
For a given regular expression, e.g., $E = b(ab)^* + c$*, the set* $\text{pos}(E) = \{1, 2, 3, 4\}$ *includes all indexes for each symbol of E in the order of appearance.*

In the next step, the regular expression is marked with positions such that every symbol is counted as a variable. The regular expression marked with positions is denoted as $\overline{E}$ *so that* $\overline{E} = b_1(a_2b_3)^* + c_4$ *in this example. The following sets construct an NFA:*

- $\text{follow}(\overline{E})$ *is a transition set for every variable that represents the set of elements followed by a specific variable. For instance,* $\text{follow}(\overline{E}, b_1) = \{a_2, \emptyset\}$ *so that successors of* b_1 *are* a_2 *and* $\emptyset$*. All follow sets have to be complete in this part of the algorithm. In a simpler form, all follow sets constitute the successor table of the automata (Table 4).*

- *Initial positions of* $L(\overline{E})$ *is the successors of* $\emptyset$ *(initial states), called the set* $\text{first}(\overline{E}) = \{b_1, c_4\}$

- *Final positions are the elements whose successors are* $\emptyset$ *(i.e.,* ε*), named as* $\text{last}(\overline{E}) = \{b_1, b_3, c_4\}$

The constructed automaton for the regex $E = b(ab)^* + c$ based on the sets is demonstrated in Figure 20.

The elements of $\mathtt{first}(\overline{E}) = \{b_1, c_4\}$ represent the states $\{1, 4\}$ following initial state 0 with actions b, c, respectively. In other words, the initial state is connected to every state in the position of $\mathtt{first}(\overline{E})$. Hence, the method always allows a single initial state.

The number of states is therefore $n + 1$ (one initial state is added) where the position set $\mathtt{pos}(\overline{E})$ has n elements in the final constructed PRA. Apparently $\mathtt{follow}(\overline{E}, x)$ for every $x \in \mathtt{pos}(\overline{E})$ is a set that represents a row in the transition matrix generated as a result of the method.

To generalize the conversion from an SRE to a PRA, the following theorem (which is an extension to the theorem in Chapter 2) is proposed by establishing the equivalence between the models.

Theorem 5.2 (Equivalence of SRE and PA)
L is a stochastic regular language if there is a stochastic regular expression (SRE) such that $L(SRE) = L$ then, if and only if there exists a probabilistic Rabin automation (a non-deterministic finite automation with probabilities) such that $L(PRA) = L$.

The following lemma will complete the proof of equivalence.

Lemma 5.2 (Existence of a PRA from an SRE)
For any stochastic regex SRE, there is a PRA such that $L(PRA_{SRE}) = L(SRE)$

The proof of the lemma is built on the Glushkov's method with a probabilistic extension. More explicitly, the elements of $\mathtt{first}(E)$ and $\mathtt{follow}(E, x)$ are adapted to the probabilistic choice. The above-mentioned example with probabilistic extension is provided in the following subsection.

5.2.1 *Glushkov's Construction with Probabilistic Extension*

Let $E = b(ab)^{*0.3}_{[4]} + c_{[6]}$ be a SRE on the alphabet $\Sigma = \{a, b, c\}$ (the probabilities and rates are arbitrary). The set $\mathtt{first}(\overline{E}) = \{b_1, c_4\}$ evolves to $\mathtt{first}_p(\overline{E}) = (b_1, 0.4), (c_4, 0.6)$ that is denoted as $\mathtt{first}_p(\overline{E})$ (we obtain 0.4 by $\frac{4}{4+6}$ and 0.6 by $\frac{6}{4+6}$). Unlike the non-probabilistic version, we have to consider the termination probability. Therefore, any Kleene star operation that has the form a^{*p} where p is a loop probability can be re-described as $a^{*p}(p) + \varepsilon(1 - p)$.

Hence, for example, the set $\mathtt{follow}(\overline{E}, b_1) = \{a\}$ evolves to $\mathtt{follow}_p(\overline{E}, b_1) = \{(a, 0.3), (\varepsilon, 0.7)\}$. As a result, the Kleene operation requires an additional state in the probabilistic case, and the elements of $\mathtt{first}(\overline{E})$ and $\mathtt{follow}(\overline{E}, x)$ evolve to (x, p) from x where $p > 0 \in \mathbb{R}$. $\mathtt{last}(\overline{E})$ evolves to $\{b_1, \varepsilon_5\}$ from $\{b_1, b_3, c_4\}$ that is denoted as $\mathtt{last}_p(\overline{E})$.

The constructed PRA is then demonstrated in Figure 21. Glushkov's construction algorithm can also be processed inductively for the regex syntax [26]. In the following proof we use a similar method including the probabilities to show that every SRE has at least one corresponding PRA. Therefore, it is constructively provided how to build these sets. For the simplicity, we use E instead of $\overline{E}$ below.

Proof.— We build a PRA for an SRE by structural induction over the SRE syntax ($\sum_i{}^G E_{i[n_i]}$ and (E) do not have an impact on the construction of PRA, hence it is not presented in the construction algorithm).

$$E := \alpha \;\Big|\; \sum_i E_{i[n_i]} \;\Big|\; \sum_i{}^G E_{i[n_i]} \;\Big|\; E_1 : E_2 \;\Big|\; E^{*f} \;\Big|\; E^{+f} \;\Big|\; (E)$$

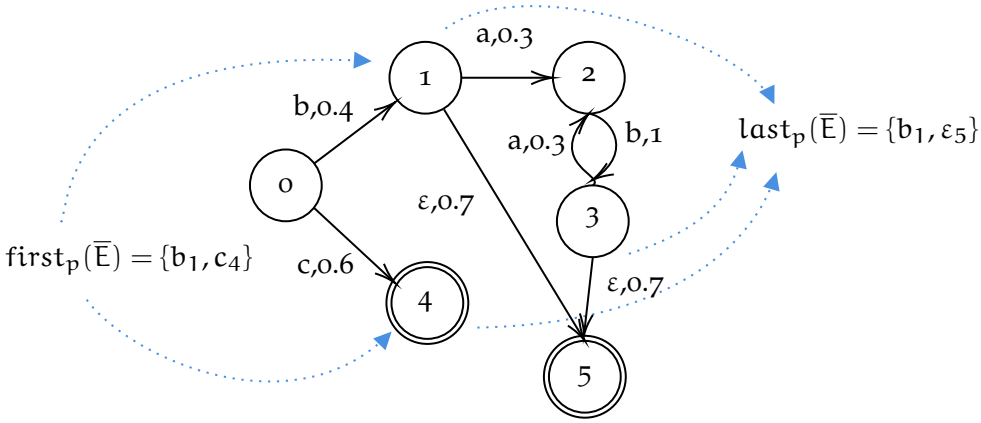


Figure 21: The PRA constructed from SRE $E = b(ab)^{*0.3}_{[4]} + c_{[6]}$

- Trivial cases : Empty set, ϵ transition, and only one transition
  - ⋆ In the case of empty set, there exists only an initial state; therefore, $pos(E)$ is empty.

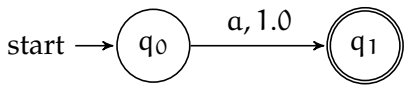
    if $E = \emptyset \implies first_p(E) = last_p(E) = follow_p(E, q_0) = \emptyset$

  - ⋆ By definition, there exists an initial state, and it is executed once with ϵ by remaining in the initial; hence, $pos(E)$ is still empty.

    If $E = \varepsilon \implies first_p(E) = last_p(E) = follow_p(E, q_0) = \emptyset$

  - ⋆ The atomic action creates only a single transition and halts such that

    If $E = a \implies first_p(E) = \{(q_1, 1.0)\}$, $last_p(E) = \{q_1\}$, and there exists no transition after a hence, $follow_p(E, q_0) = \emptyset$.

- Case Choice $E = \sum_i E_i[n_i]$: By induction hypothesis,
  there are PRA_{SRE_1}, PRA_{SRE_2},...,PRA_{SRE_n} such that, $L(PRA_{SRE_1}) = L(E_1)$, $L(PRA_{SRE_2}) = L(E_2)$, ..., $L(PRA_{SRE_n}) = L(E_n)$.

  We construct the PRA for $E = \sum_i E_i[n_i]$ where n_i is the choice rate of SRE and $n_i \in \mathbb{N}^+$. By definition, there exists a single initial state in Glushkov automata. Therefore, this rule has to be fulfilled during the union. Inserting a new state and creating outgoing transitions with the given probability ($\frac{n_i}{\sum n_i}$) for every element) to every initial state of the union elements would be the easiest solution. However, such a method creates ε transitions which we avoid in the Glushkov's construction. Instead, we get the union of all the initial states and multiply the probabilities of every outgoing transition with the relevant probability.

$$\mathrm{first}_p(E_{1[n_1]} + E_{2[n_2]}) = \mathrm{first}_p(E_1) \cup \mathrm{first}_p(E_2)$$

Multiply the probability of all elements by $\frac{n_1}{n_1+n_2}$ and $\frac{n_2}{n_1+n_2}$

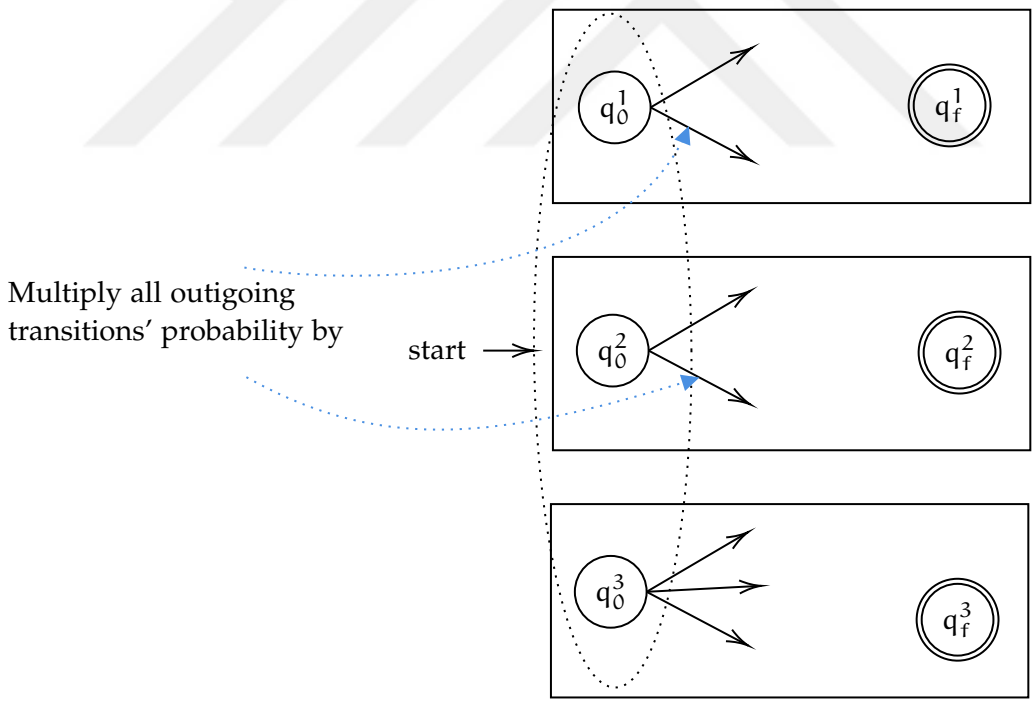
The final states are combined.

$$\mathrm{last}_p(E_1[n_1] + E_2[n_2]) = \mathrm{last}_p(E_1) \cup \mathrm{last}_p(E_2) \tag{94}$$

The relation of states are obtained by multiplying the probability in which the elements of the choice operation are included.

$$\mathrm{follow}_p(E_1[n_1] + E_2[n_2], q_i) = \begin{cases} \mathrm{follow}_p(E_1, q_i) & \text{if } i \in \mathrm{pos}(E_1) \\ \mathrm{follow}_p(E_2, q_i) & \text{if } i \in \mathrm{pos}(E_2) \end{cases} \tag{95}$$

At this point, the probability of all elements of $\mathrm{follow}_p(E_1, q_i)$ and $\mathrm{follow}_p(E_2, q_i)$ are multiplied by $\frac{n_1}{n_1+n_2}$ and $\frac{n_2}{n_1+n_2}$, respectively.



- Case Concatenation $E_1 : E_2$: We have an induction hypothesis for two operands of the concatenation ($E = E_1 : E_2$); thus, there is a PA_{E_1} for E_1, and PA_{E_2} for PA_{E_2} such that $L(PA_{E_1}) = L(E_1)$ and $L(PA_{E_2}) = L(E_2)$. The construction is achieved as follows to show that there exists PA_E for E such that $L(PA_E) = L(E)$:

Initial states will be the first operand's (E_1) initial states unless it produces ϵ. Otherwise, they will be the union of two operands' initial states.

$$\mathrm{first}_p(E_1 : E_2) = \begin{cases} \mathrm{first}_p(E_1) & \text{if } \varepsilon \notin L(E_1) \\ \mathrm{first}_p(E_1) \cup \mathrm{first}_p(E_2) & \text{if } \varepsilon \in L(E_1) \end{cases} \tag{96}$$

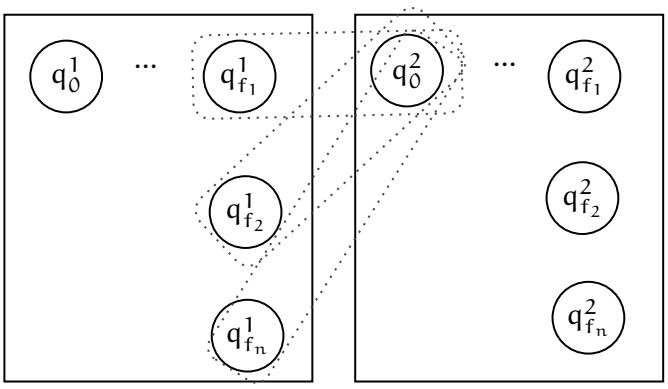
In the second case, probabilities of the elements of $\mathrm{first}_p(E_2)$ are multiplied by the probability of ε.

Similarly, final states are obtained by the second operand's final states unless ε is second operand's element.

$$\mathrm{last}_p(E_1 : E_2) = \begin{cases} \mathrm{last}_p(E_2) & \text{if } \epsilon \notin L(E_2) \\ \mathrm{last}_p(E_1) \cup \mathrm{last}_p(E_2) & \text{if } \epsilon \in L(E_2) \end{cases} \tag{97}$$

Any state (q) will be concatenated (with 1.0 probability) to initial states of E_2 if it is the final of E_1.

$$\mathrm{follow}_p(E_1 : E_2, q_i) = \begin{cases} \mathrm{follow}_p(E_1, q_i) & \text{if } q_i \in \mathrm{last}(E_1) \\ \mathrm{follow}_p(E_1, q_i) & \\ \cup\{(y, 1.0) \mid \forall y \in \mathrm{first}_p(E_2)\} & \text{if } q_i \in \mathrm{last}_p(E_1) \\ \mathrm{follow}_p(E_2, q_i) & \text{if } i \in \mathrm{pos}(E_2) \end{cases} \tag{98}$$



- Case Plus and Kleene Closure E^{+f}/E^{*f}: The initial state of an SRE E will remain the same when the plus and Kleene closure operation are applied where f is the loop probability.

$$\mathrm{first}_p(E) = \mathrm{first}_p(E^{+f}) = \mathrm{first}_p(E^{*f}) \tag{99}$$

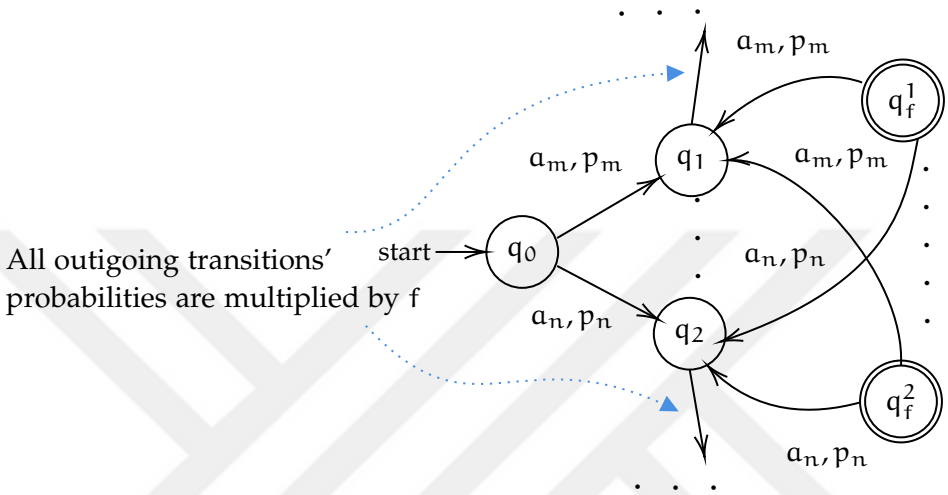
The final states alter since we add the termination transition and the probability.

$$\mathrm{last}_p(E^{+f}) = \mathrm{last}_p(E^{*f}) = \bigcup(q_i, p) \text{ if } \varepsilon \in \mathrm{follow}_p(E, q_i). \tag{100}$$

The key step is to construct the follow_p sets by creating transitions from every final state to initial states where the loop starts.

$$\text{follow}_p(E^{+f}, x) = \begin{cases} \text{follow}_p(E, q_i) & \text{if } q_i \in \text{first}_p(E) \\ \cup\{(q,p) | \forall q \in \text{first}_p(E)\} & \text{if } q_i \in \text{last}_p(E) \end{cases}$$
$$\text{where } p \in \mathbb{R}, \text{ and the probability of the action} \tag{101}$$

The probability of all elements of $\text{follow}_p(E, q_i)$ are multiplied by f if $q_i \in \text{first}_p(E)$.



To complete our hypothesis, we show that the probability of string matching are equal for the same execution path on the equivalent SRE and PRA in theorem 5.3.

Theorem 5.3 (Equivalance of probabilistic matching for an SRE and the constructed PRA) *Let E be a Stochastic Regular Expression (SRE). Let PRA_E be the PA that is acquired by applying above algorithm to E. Then,*

$$\forall s \in \Sigma^* : [\![E]\!]s = [\![\text{PRA}_E]\!]s \textit{ where } s = s_1 s_2 \cdots s_n.$$

holds.

Proof. — For every case by induction:

- The atomic case is trivial. Let $E = a$. Then $\text{PRA}_E = (\Sigma, \{q_0, q_1\}, q_0, P, \{q_1\})$ with $P(q_0, q_1, a) = 1$.

$$[\![E]\!]s = 1 = [\![\text{PRA}_E]\!]s \text{ if } s = a$$
$$[\![E]\!]s = 0 = [\![\text{PRA}_E]\!]s \text{ if } s \neq a$$

- Let $E = E_1 : E_2$ and $[\![E_i]\!]s = [\![\text{PRA}_{E_i}]\!]s$ by induction.

$$\begin{aligned} [\![E]\!]s &= \sum_{i=1}^{n} [\![E_1]\!]s_1 \ldots s_i \cdot [\![E_2]\!]s_{i+1} \ldots s_n \\ &= \sum_{i=1}^{n} [\![\text{PRA}_{E_1}]\!]s_1 \ldots s_i \cdot [\![\text{PRA}_{E_2}]\!]s_{i+1} \ldots s_n \\ &= \sum_{i=1}^{n} \left(\sum_{\sigma \in \text{PRA}_{E_1}} \prod_{j=1}^{|\sigma|} P(q_j, q_{j+1}, s_i) \cdot \sum_{\sigma' \in \text{PRA}_{E_2}} \prod_{j=1}^{|\sigma'|} P(q_j, q_{j+1}, s_{i+1}) \right) \end{aligned}$$

we multiply the probability for each path (σ) in the first part of the PRA with the probability for each path (σ') in the second part of the PRA to unify the sum over the paths. Since every path in the first part ends with a state in F_1, every path in the second part begins with q_0^2, and every state in F_1 has a transition to every state that is included in $\mathtt{first}_p(E_2)$ followed by q_0^2 with probability 1.0.

In other words, the state q_0^2 is discarded, and the two PRAs get connected. One path is described for each combination of those paths from the two partial PRA as follows: $\sigma^E = \{\sigma_1, a, \sigma_2 | \sigma_1 \in PRA_{E_1}, \sigma_2 \in PRA_{E_2}\}$.

Furthermore, $\forall q_f \in F_1$, and we have $P(q_f, q_1, a) = 1$; hence, this transition is obtained by multiplying with 1.0 without altering the result. a is the corresponding transition label between the initial state and the states in the set $\mathtt{first}_p(E_2)$.

$$[\![E]\!]s = \sum_{\sigma \in \sigma^E} \prod_{i=1}^{|\sigma|} P(q_i, q_{i+1}, a) = [\![PRA_E]\!]s$$

- Let $E = (E')^{*\frac{i}{j}}$ (the fraction representation of the loop probability, with $i, j \in \mathbb{N}_+$) and $[\![E']\!]s = [\![PRA_{E'}]\!]s$ by induction.

$$\begin{aligned}[\![E]\!]s &= [\![\epsilon[j-i] + E' : E[i]]\!]s \\ &= \frac{j-i}{j}[\![\epsilon]\!]s + \frac{i}{j}[\![E' : E]\!]s\end{aligned}$$

We get an infinite sum by recursively applying it.

$$[\![E]\!]s = \frac{j-i}{j} \cdot \sum_{k=0}^{\infty} \frac{i^k}{j^k}[\![E'^k]\!]s$$

We obtain the following equation by using the induction hypothesis with E^i being the concatenation of E with itself i times

$$\begin{aligned}[\![E]\!]s &= \frac{j-i}{j} \cdot \left([\![\epsilon]\!]s + \sum_{k=1}^{\infty} \frac{i^k}{j^k} \overbrace{[\![PRA(E')^k]\!]s}^{[\![E'^k]\!]s} \right) \\ &= \frac{j-i}{j} \cdot \left([\![\epsilon]\!]s + \sum_{k=1}^{\infty} \frac{i^k}{j^k} \sum_{\sigma \in \sigma^k} \prod_{m=1}^{|\sigma|} P(q_m, q_{m+1}, s_m) \right)\end{aligned}$$

. This transition is added k times to the path with $\sigma^k = \{q_0 \epsilon \sigma^{k\prime} | \sigma^{k\prime}\}$ and $\sigma^{k\prime} = \{\sigma^1 \epsilon q_0 \epsilon \sigma^2 \epsilon q_0 \epsilon \sigma^3 \ldots \sigma^k | \sigma^i \in PRA(E')\}$ being a path concatenation of k paths through $PRA(E')$, which is used for looping and additionally starting at q_0 instead of q_0', when $k \to \infty$; therefore, $\frac{i^k}{j^k}$ converges to $\frac{j}{j-i}$.

$$[\![E]\!]s = \frac{j-i}{j} \cdot \frac{j}{j-i} \cdot \left([\![\epsilon]\!]s + \sum_{\sigma \in \sigma^k} \prod_{m=1}^{|\sigma|} P(q_m, q_{m+1}, s_m) \right)$$

Now, we construct σ^E as the set of all paths through PRA_E: $\sigma^E = \{\sigma^k \epsilon q_0 \epsilon q_f | \sigma^k \in , k \in \mathbb{N}_+\} \cup \{q_0 \epsilon q_f\}$.

$$[\![E]\!]s = \sum_{\sigma \in \sigma^E} \frac{\prod_{m=1}^{|\sigma|} P(q_m, q_{m+1}, s_m)}{P(q_0, q_f, \epsilon)}$$

The last path is only relevant if $s = \epsilon$ and, therefore, corresponds to the first term.

$$[\![E]\!]s = \sum_{\sigma} \prod_{m=1}^{|\sigma|} P(q_m, q_{m+1}, s_m) = [\![PRA_E]\!]s$$

- Let $E = \sum_i E_{i[n_i]}$ and $[\![E_i]\!]s = [\![A(E_i)]\!]s$ by induction.

$$\begin{aligned}
[\![E]\!]s &= \sum_k \frac{n_k}{\sum_j n_j} \cdot [\![E_k]\!]s \\
&= \sum_k \frac{n_k}{\sum_j n_j} \cdot [\![PRA(E_k)]\!]s \\
&= \sum_k \frac{n_k}{\sum_j n_j} \cdot \sum_{p \in a(E_k)} \prod_{i=1}^{|p|} P(q_i, q_{i+1}, s_i) \\
&= \sum_k \sum_{p \in PRA(E_k)} \frac{n_k}{\sum_j n_j} \cdot \prod_{i=1}^{|p|} P(q_i, q_{i+1}, s_i)
\end{aligned}$$

By construction, we can substitute the fraction:

$$[\![E]\!]s = \sum_k \sum_p P(q_0, q_0, \epsilon) \cdot \prod_{i=1}^{|p|} P(q_i, q_{i+1}, s_i)$$

The path that is chosen to start at q_0 is changed with q_0^k, the initial state of $PRA(E_k)$. It integrates the first factor into the product, and the path $\sigma^E = \{q_0 \epsilon \sigma^k | \sigma^k \in PRA(E_k)\}$ is chosen globally.

$$[\![E]\!]s = \sum_{\sigma \in \sigma^E} \prod_{i=1}^{|\sigma|} P(q_i, q_{i+1}, s_i) = [\![PRA_E]\!]s$$

□

5.2.2 *Illustrative Example: From an SRE to a PA*

Let us consider the same example (provided in (92)) to run the transformation process from the an SRE to a PRA. Let E be a stochastic regular expression defined on the alphabet $\Sigma = \{start, login, authenticationFail, logout, sendMsg, msgFail, terminate, success, sendMsg\}$.

$$x_0 = \mathrm{login} : \big(\mathrm{sendMsg} : \big(\mathrm{success} : ((\mathrm{sendMsg} : \mathrm{success})^{*0.25} \\ : (\mathrm{success} : \mathrm{logout} : \mathrm{terminate}))_{[95]} \\ + \mathrm{msgFail}_{[5]}\big)_{[65]} + \mathrm{logout} : \mathrm{terminate}_{[20])} + \mathrm{authenticationFail}_{[15]}\big) \tag{102}$$

According to Glushkov's construction, initially we **index** the actions.

$$\overline{E} = \mathrm{login}_1 : (\mathrm{sendMsg}_2 : (((\mathrm{success}_3 : \mathrm{sendMsg}_4)^{*0.25} \\ : \mathrm{success}_5 : \mathrm{logout}_6 : \mathrm{terminate}_7)[95] + \mathrm{msgFail}_8[5])[65] \\ + \mathrm{logout}_9 : \mathrm{terminate}_{10}[20] + \mathrm{authenticationFail}_{11}[15]) \tag{103}$$

After that we start the construction of the Glushkov sets.

$$\overline{\mathrm{first}}(\overline{E}) = \{(\mathrm{login}_1, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{login}_1) = \{(\mathrm{sendMsg}_2, 0.65), (\mathrm{logout}_9, 0.2), (\mathrm{authenticationFail}_{11}, 0.15)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{sendMsg}_2) = \{(\mathrm{success}_3, 0.95), (\mathrm{msgFail}_8, 0.5)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{logout}_9) = \{(\mathrm{terminate}_{10}, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{authenticationFail}_{11}) = \emptyset$$
$$\mathrm{last}(\overline{E}) = \{\mathrm{authenticationFail}_{11}\}$$

The first snapshot of the partially constructed probabilistic automata is demonstrated in Figure 22.

Snapshot 1

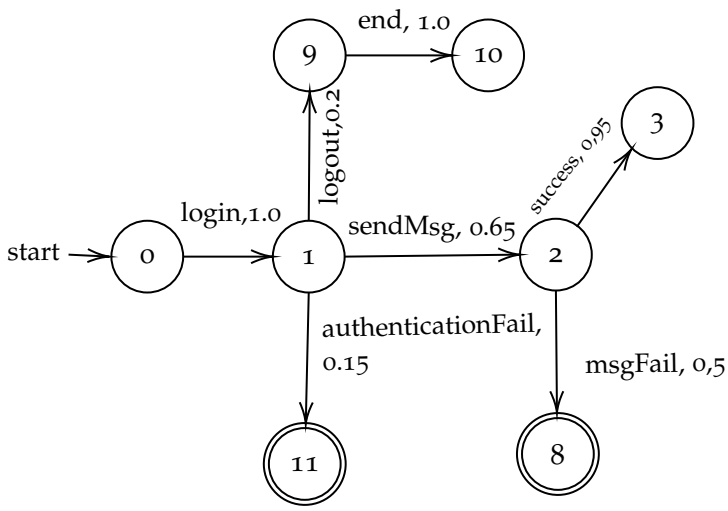


Figure 22: Partially constructed PRA of example 19: $\mathrm{Snapshot}_1$

The final constructed PRA (Figure 23) is equivalent to the initial automaton provided in Figure 19, although the constructed PRA includes more states since it is not minimized. In this thesis, our focus is to prove that the conversion between an SRE

and a PRA is possible and our approach is correct. Therefore, the state equivalence, minimization, and probabilistic bisimulation relations [82] are out of scope.

$$\overline{\mathrm{first}}(\overline{E}) = \{(\mathrm{login}_1, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{login}_1) = \{(\mathrm{sendMsg}_2, 0.65), (\mathrm{logout}_9, 0.2), (\mathrm{authenticationFail}_{11}, 0.15)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{sendMsg}_2) = \{(\mathrm{success}_3, 0.95), (\mathrm{msgFail}_8, 0.5)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{logout}_9) = \{(\mathrm{terminate}_{10}, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{authenticationFail}_{11}) = \emptyset$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{login}_1) = \{(\mathrm{sendMsg}_2, 0.65), (\mathrm{logout}_9, 0.2), (\mathrm{authenticationFail}_{11}, 0.15)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{success}_3) = \{(\mathrm{sendMsg}_4, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{success}_5) = \{(\mathrm{logout}_6, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{msgFail}_8) = \emptyset$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{sendMsg}_4) = \{(\mathrm{success}_5, 0.75), (\mathrm{success}_3, 0.25)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{logout}_6) = \{(\mathrm{terminate}_7, 1.0)\}$$
$$\overline{\mathrm{follow}}(\overline{E}, \mathrm{terminate}_7) = \emptyset$$
$$\mathrm{last}(\overline{E}) = \{\mathrm{authenticationFail}_{10}, \mathrm{msgFail}_8, \mathrm{terminate}_7\}$$

Snapshot 2

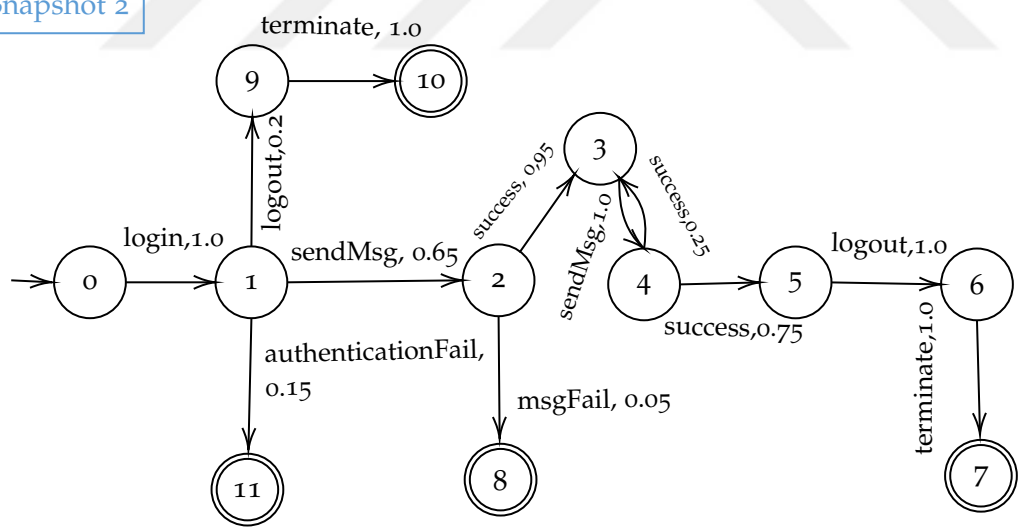


Figure 23: The final constructed PRA of example 19: $\mathrm{Snapshot}_2$

5.3 CONCLUSION

In this chapter, we have established a translation relation between PRA and SRE models with theorems and examples. This bidirectional translation shows that PRA and SRE models are identical. We have provided a probabilistic extensions of well-established methods, such as state elimination and regex equation solving to transform PRAs into SREs and executed them on some examples.

State elimination algorithm complexity is quadratic in time; nevertheless, the size of the regular expression can explode [69] as a well-known explosion problem in automata theory. Among various strategies, there is one, for example, that is developed using heuristic in state selection [118]. Similar heuristics can be applied in our approach.

We have implemented a probabilistic version of Glushkov's construction and generalized its theorem to transform an SRE into a PRA. This transformation generates PRA in the size of SREs; however, the obtained automata is still not minimized. There, some techniques can be applied using probabilistic bisimulation [86], or state equivalence techniques [50] to minimize the PRA.

Part III

INCREMENTAL QUANTITATIVE VERIFICATION

6 INCREMENTAL QUANTITATIVE VERIFICATION

This section introduces an incremental quantitative verification framework using the formalism described in the previous chapter 4. Initially, we clearly state the motivation why we need a new formalism instead of using an existing quantitative model checking frameworks.

6.1 MOTIVATION: USING STOCHASTIC REGULAR EXPRESSION FORMALISM FOR INCREMENTAL VERIFICATION

Incremental techniques adopt in two styles: (1) *reusing partial calculations iteratively to empower the computation* e.g., dynamic programming, counter example guided approaches, and (2) *reusing computational results for different versions of models*. In this thesis, the focus is more on the latter case; the incremental calculation especially for the modified models, in other words versions of the models. The versions of the systems, models or the system code usually bear small changes in the iterative development of the systems [64].

On the other side, such changes can create big effects on the probabilistic models and force to re-verify the whole system to maintain the requirements' fullfillness. Additionally, probabilistic systems are usually modeled with an automaton together with probabilistic data. Examples to probabilistic models are Markov chains, Markov decision processes [9], and stochastic Petri nets [75]. Such graph-based models can be represented in a matrix form to solve equations based on the states and the transition probabilities in the process of a probabilistic analysis (a detailed classification of stochastic process algebra can be found in [30]). Nevertheless, such models lack the expressiveness to identify the change effect on the model.

Let us clarify this argument with the example provided in Figure 24. When a change is introduced, for instance, the transition a_3 is removed from q_3 to q_2; Instead, it is added to a newly created final state q_{f_3} (as shown in dotted area). More precisely, the three change operations are listed as a deletion of one transition (q_3, a_3, q_2), and addition of one transition (q_3, a_3, q_{f_3}) and addition of one state q_{f_3}. In other words, defining edit operations or change operations consists of only updating states and transitions.

Nevertheless, the structure of the model evolves from a loop to a choice path for the state q_3 within this change. Our approach lies upon the idea how to localize the change more expressively. Thus, we propose applying edit operations using the mathematical verification framework of SREs where we can represent the same change as a deletion of a loop and addition of a choice operation.

In specific, the model has two choices from q_0, which can be shown as

$q_0 = E_{1[3]} + E_{2[7]}$ where E_1 represents the path $a_1 : a_5$ and,
$E_2 = a_0 : a_2 : (a_2 a_3)^{*0.2} : a_4$.

In this structure, a change can be applied on the part of E_2, that is deletion of a loop $(a_2 a_3)^{*0.2}$, and instead, addition of a of a choice $a_{4[6]} + a_{3[4]}$. Therefore, the updated E_2 would look like

$$E_2 = a_0 : a_2 : (a_{4[8]} + a_{3[2]})$$

As a consequence, this change will only affect the model partially and enable the localization of the change. At this point, the reader would come up with some questions for the position of the change, the density of the model, etc. Not only do we discuss different cases of the changes and their impacts on the model, but also precise edit operations to realize these changes are presented in the following sections where the details of the quantitative incremental verification are given.

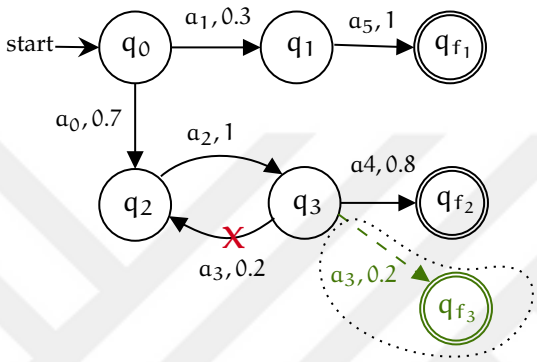


Figure 24: Changing probabilistic model

6.2 INCREMENTAL QUANTITATIVE VERIFICATION (IQON) FRAMEWORK

In this section, the incremental verification for probabilistic systems is discussed. As we mention in the previous section, we focus on the incremental calculations especially for the modified models, in other words versions of the models.

Incrementality and modularity in this study lies upon the idea of the compositional functions on the tree operations, which are formally set up in Chapter 4. In this regard, tree operations are a means to manipulate and apply some changes in the model.

The incremental setup and the overall approach are presented in Figure 25. The input model of an incremental verification framework is a stochastic regular expression (SRE). However, the initial model can also be a probabilistic Rabin automation obtained from ampter type of abstract models, such as reactive modules [5] or process algebra [30]. Such probabilistic automata models are converted to stochastic regular expressions using various techniques such as state elimination[40] or equation solving [119]. We have discussed the details in Chapter 5 how the transformations have been achieved as well as the equivalence between models, sizes of the models, and the transformation cost.

The probabilistic requirements are described via the PACTL logic. After parsing the initial model, an SRE tree is constructed via AST and attached with the PACTL attributes (part 1 in Figure 25). Any change detected from logs is applied as an SRE edit operation, and the SRE tree is updated for probabilistic results (part 2 in Figure 25). From the viewpoint of software engineering, we first clarify the terms of *change*, *evolution step*, *version*, and *edit operation*.

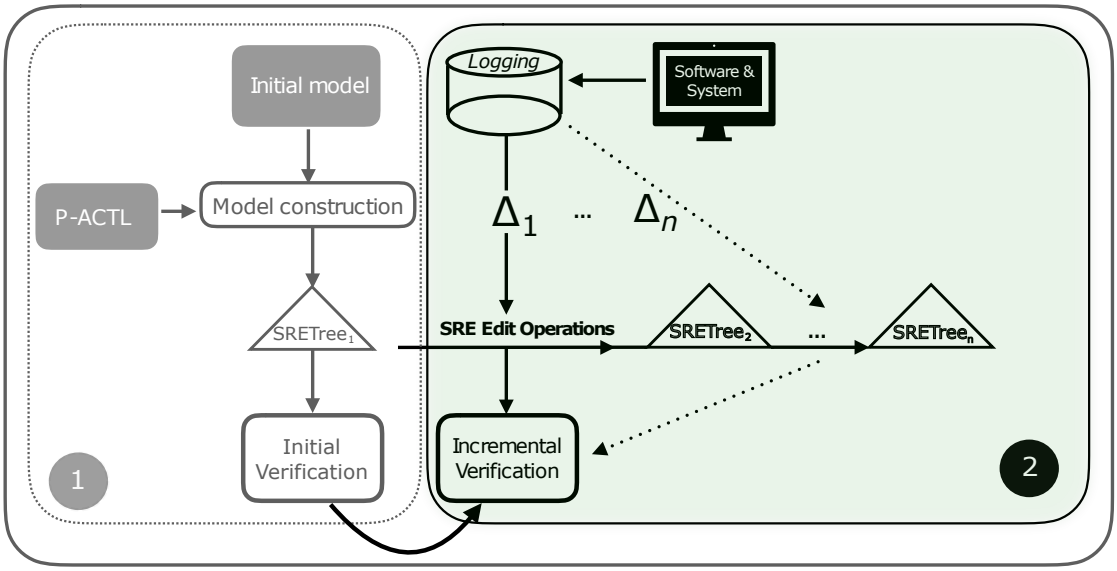


Figure 25: *IQon* framework

Definition 6.1 (Evolution step)
An evolution step is actually a snapshot of the system at any given time; in other words, a version of the system.

Definition 6.2 (Change)
A change, which is specified between two evolution steps, is a collection of edit operations that transform a model into another version of the same type of the model and is denoted as Δ.

A change is also called a *difference*, and the calculation of a difference between two versions of the models is done with two methods in the literature: (1) **operation-based differencing**, which is also known as "logging", and (2) **state-based** comparison [93]. We refer to the paper [95] for the comparison of these techniques. In the both definitions, a *change* converts a model into another version through an evolution step.

Definition 6.3 (Edit operation)
A change is a difference between two versions of models, which is defined by a set of edit operations. The set of edit operations has to be complete to apply any change in the model. An edit operation can be atomic or complex composed of multiple edit operations. ***Atomic edit operations*** *cannot be split into smaller operations, i.e., creation and deletion of nodes and edges of a graph or tree-structured model. Any change must be representable by atomic edit operations.* ***Complex edit operations*** *can be split into multiple atomic actions.*

Let M_1 and M_2 be instances of the model $\mathbb{M}$. $\Delta_{1\to 2}$ represents the *change* via edit operations between models M_1 and M_2. We entitle 1 and 2 as evolution steps.

In our *IQon* framework, a system and the requirements are specified by an SRE and a PACTL formula. Therefore, the language of the SRE and PACTL are processed by parsing and constructing an SRE tree out of the abstract syntax tree (AST) [70], carrying the computational results of the PACTL model checking whose formal foundations are provided in Chapter 4. The initial construction is demonstrated in Figure 25 in the left dotted box (part 1).

We use shift-reduce parsing [4] that generates an AST with a bottom up approach for the initial construction reported in Algorithm 4. Each node of the AST has its

value computed from the PACTL model checking algorithm (Chapter 4) by function `contructSREnodewithMCValues(N)`.

```
while topTerm(Stack) ≠ '#' and input ≠ '#' do
    p = precedence [topTerm, nextToken] ;
    if p == "<" or p == "=" then
        ;                                                    /* shift */
        shift nextToken onto stack and advance input ;
    else if p == ">" then
        ;                                                    /* reduce */
        find the shallowest pair of terminals d and s on the stack such that d <
         s,where d is the deeper terminal ;
        pop everything above d off the stack ;
        push N, the general non-terminal, onto the stack ;
        E ← contructSREnodewithMCValues(N) ;
        add E to tree T_E ;
    else
        in other case
    end
    E_root ← E
end
```

Algorithm 4: Bottom up parsing [4] with the construction of additional model checking values

The construction of the SRE tree is provided in Algorithm 5, and the constructed SRE tree is formally defined as

- $T_E = \{E_{root}, \mathcal{E}, \Sigma)\}$, where E_{root} is the root node,
- $\mathcal{E}$ is the finite set of all nodes, and
- Σ is the alphabet.

An SRE node in the SRE tree can have four types, such as, *Action*, *Choice*, *Concatenation*, and *Kleene star*, and denoted as $_{<\alpha>}E$, $_{<choice>}E$, $_{<concat>}E$ and $_{<kleene>}E$, respectively. Every node type is a node E, and the calculations for the model checking are processed until the root node. Hence, verifying the root node E_{root}, which can be any node type, will result in verifying the system. The calculations of the SRE values that correspond to the marked lines (1),(2),(3), and (4) in Algorithm 5 are previously explained in Chapter 4 with model checking algorithms.

When a change occurs, the SRE tree is updated only for the part that is subject to the change, which is achieved via a set of *edit operations* in the SRE tree. We provide a complete set of *edit operations* over the SRE syntax in the following subsection.

6.2.1 *SRE Edit Operations for Incremental Analysis*

Up to this point, we describe the initial construction of an SRE model as a tree, which corresponds to the *part 1* of the *IQon* framework in Figure 25. Once we parse the SRE model, we can apply manipulations on the constructed tree. Such manipulations are achieved by means of a complete list of edit operations over the SRE syntax, which

```
contructSREnodewithMCValues(N)
Data: Non-terminal Nx
Result: Constructed SRE node E
if N is an "action' rule then                                  /* E = α */
    create action node <α>E;
(1) compute initial and attach the hashmap to <α>E;
    E ← <α>E ;
else if N is a "choice" rule then                              /* E = E1[n1] + E2[n2] */
    create choice node <choice>E ;
(2) compute choice and attach the hashmap to <choice>E ;
    set parent of E1 as <choice>E;
    set parent of E2 as <choice>E;
    E ← <choice>E ;
else if N is a "concat" rule then                              /* E = E1 : E2 */
    create concatenation node <concat>E;
(3) compute concatenation and attach the hashmap to <concat>E;
    set parent of E1 as <concat>E;
    set parent of E2 as <concat>E;
    E ← <concat>E;
else if E is a "Kleene" rule then                              /* E' = E*f */
    create Kleene node <kleene>E;
(4) compute Kleene and attach the hashmap to <kleene>E;
    set parent of E as <kleene>E;
    E ← <kleene>E ;
return E
```

Algorithm 5: Algorithm `constructSREnodewithMCValues`-revisited from Chapter 4

corresponds to the *part 2* of the *IQon* framework. Let us recall the syntax of an SRE E over an alphabet Σ.

$$E := \alpha \left| \sum_i E_{i[n_i]} \right| \sum_i^G E_{i[n_i]} \left| E_1 : E_2 \right| E^{*f} \left| E^{+f} \right| (E) \tag{104}$$

with $\alpha \in \Sigma \cup \{\varepsilon\}$, $n_i \in \mathbb{Z}^+$, $f \in [0, 1] \subset \mathbb{R}$, and every term E_i is an SRE.

All possible edit operations are defined for every SRE term. In the following list of edit operations, we denote *addition, deletion, replacement*, and *update* using the symbols $+$, $-$, $\rightleftarrows$, and $\circlearrowright$, respectively.

An *action* can be added/created or deleted, and replaced with another action. The alphabet Σ needs to be updated eventually. Possible edit operations for an *action* term are:

- $[\alpha, -]$ deletion of an action node,
- $[\alpha, +]$ addition of an action node,
- $[\alpha, \alpha', \rightleftarrows]$ replacement of an action node.

The *choice* term $\sum_i E_{i[n_i]}$ can include sub-terms E_i from 1 up to n; every term E_i can be added, deleted, or replaced. Additionally the rate n_i can be updated with $n_i{}'$, where n_i and $n_i{}' \in \mathbb{N}$. As long as $i \geqslant 2$, these edit operations can be applied. Otherwise, E is

not a summation term. In case of $i < 2$, then the choice term is removed, and its sub-term is made connected to the parent term of E. Possible edit operations for a *choice* term are:

- $[E, i, -]$ deletion of a choice sub-term (a simple delete operation of the choice term is demonstrated in Figure 26),
- $[E, i, +]$ addition of a choice sub-term,
- $[E, i, E_i{}', \rightleftarrows]$ replacement of a sub-term,
- $[E, i, , n_i, n_i{}', \circlearrowright]$ updating the rate of a sub-term.

The *guarded choice* term $\sum_i{}^G$ includes the same edit operations as *choice* edit operations only with additional constraints during the sub-term addition. The constraint is preserving the uniqueness of the prefixes that belonging the sub-terms or the uniqueness of themselves in case they are single *actions*.

The concatenation term $E_1 : E_2$ can include two sub-terms, every term E_1, or E_2 can be added, deleted or replaced. Since the concatenation is not commutative, the order of the terms is important. Hence, replacing E_1 and E_2 is an edit operation. As long as $i \geqslant 2$, these edit operations can be applied. Otherwise, the concatenation term E is removed, and its sub-node is made connected to the parent node of E. Possible edit operations for a *concatenation* term are:

- $[E, i, -]$ deletion of a concatenation sub-term,
- $[E, i]$ addition of a concatenation sub-term,
- $[E, i, E', \rightleftarrows]$ replacement of a sub-term,
- $[E, i, E, j, \rightleftarrows]$ changing the order of concatenation sub-terms, where i and j represent the order.

The Kleene and Plus closure term E^{*f} or E^{+f} has one term E and the loop probability. The term can be added, deleted, and replaced. The loop probability can be updated as long as $f \in [0, 1] \subset \mathbb{R}$. The other edit operation removes the f, that is, removing the loop completely and leaving the term as E. Possible edit operations for *closure* terms are:

- $[E, -]$ deletion of the Kleene (or Plus) closure sub-term
- $[E, f, +]$ addition of the Kleene (or Plus) closure sub-term
- $[E, E', \rightleftarrows]$ replacement of the Kleene (or Plus) closure sub-term
- $[E, f, \circlearrowright]$ updating the loop probability
- $[E, f, -]$ removing the Kleene (or Plus) closure and transforming it to E.

Any E term can either be encapsulated with parentheses, or the parentheses can be removed. Nevertheless, the verification value is not affected.

- $[E, , (), -]$ removal of parentheses,
- $[E, (), +]$ addition of parentheses.

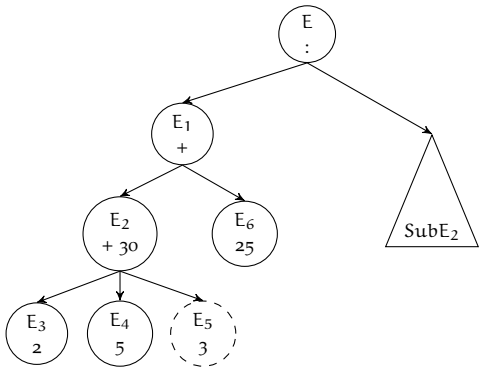


Figure 26: An illustration of a simple delete operation for the node E_5

Definition 6.4 (Delta formalism)
A change Δ between two versions of the model is a set of edit operations applied sequentially; $\Delta = \{Op_1, Op_2, \ldots Op_n\}$. An operation Op gets the id of E on which the change is applied. The other parameters are defined from the edit operations catalogue given above (e.g., $Op_i = [E_{id}, E, j, -]$).

All basic change operations lead the *update tree* operation to recursively propagate the change until the top element. The cost of the *update* operation obviously depends on the change location, and its level on the tree, in other words the depth of the node where the change occurs, and relies on the number of nodes in an unbalanced tree in the worst case.

The incremental verification Algorithm (6) gets the Δ as an input in a list of edit operations (the order of edit operations is substantial since the execution of edit operations might lead different SRE trees in the final result). Edit operations are applied based on the node ids and processed sequentially. Then, the tree is recursively updated through the parent nodes. The compositional probabilistic functions are applied on each node based on their types formalized in Chapter 4 (Algorithm 6).

Model check $(E, \Delta = \{(E_{id}, Op), .., (E_{id}, Op)\})$
Data: An SRE tree T_E, target action sequence s, Δ
Result: The reachability probability for a given action/action sequence s
{
foreach (E_{id}, Op) **do**
| search(E_{id}) apply (E_{id}, p)
end
return E
}
apply (E_{id}, op) {
T_E=`contructSREnodewithValues(`E_{id}`);`
}
updateTree(T_E, E_{id})

Algorithm 6: Incremental model checking algorithm

The algorithm complexity is identified with the depth of the node from which the tree is updated together with the string calculation. In the worst case, the whole tree is visited. We cache string calculations and benefit from them during the recalculation of the nodes for efficiency reasons.

$$
\begin{aligned}
&T_E = (E_{root}, \mathcal{E}, \Sigma)\\
&E_{root} = S\\
&\mathcal{E} = \{S, S_1, S_2, E_1, E_2, E_3, E_4, E_5, E_6, E_7\}\\
&\Sigma = \{\mathtt{start}, \mathtt{login}, \mathtt{authenticationFail}, \mathtt{logout}, \mathtt{sendMsg},\\
&\mathtt{msgFail}, \mathtt{terminate}, \mathtt{success}\}\\
&\mathbf{S} = S_1 : S_2\\
&S_1 = \mathtt{start} : \mathtt{login}\\
&\mathbf{S_2} = E_{1\,[65]} + E_{5\,[20]} + \mathtt{authenticationFail}_{[15]}\\
&\mathbf{E_1} = \mathtt{sendMsg} : E_2\\
&E_2 = E_{3\,[95]} + \mathtt{msgFail}_{[5]} \xleftarrow{\mathtt{update}} \mathbf{E_2} = E_{3\,[95]} + E_{9\,[5]}\\
&E_8 = (\mathtt{msgFail} : \mathtt{repair})\\
&E_9 = {E_8}^{*0.75}\\
&E_3 = E_4 : E_7\\
&E_4 = E_6^{*0.25}\\
&E_5 = \mathtt{logout} : \mathtt{terminate}\\
&E_6 = \mathtt{success} : \mathtt{sendMsg}\\
&E_7 = \mathtt{success} : E_5
\end{aligned}
$$

Figure 27: Changing SRE model

Example 6.1 (Defining Δ in Example 27 by SRE edit operations)
We illustrate a change on the same example system (Figure 15) from Chapter 4 to demonstrate the modularity of the model checking SRE tree. Let us recall the model that is composed of two sub-components, Service 1(S_1) and Service 2(S_2), aiming to achieve a message protocol (Figure 27).

Let us assume that a change occurs in $\mathtt{Service}_2$ *by removing "msgFail" action and replacing it with "msgFail : repair" actions in a loop with the probability 0.75 aiming to decrease the probability of "msgFail" by adding a "repair" action. This change creates a new SRE* E_8 *, which leads to the replacement of "msgFail" in* E_2*. The change is propagated in the SRE tree up to the root node. The newly added nodes and the changes nodes are coloured as green in the SRE model (Figure 27). The change in terms of edit operations are described as follows:*

$$
\begin{aligned}
\Delta = \Big\{&[\mathtt{repair}, +],\\
&(E_8, [\mathtt{msgFail}, 1, +]),\\
&(E_8, [\mathtt{repair}, 2, +]),\\
&(E_9, [E_8, 0.75, +]),\\
&(E_2, [\mathtt{msgFail}, 2, E_9, \rightleftarrows])\Big\}
\end{aligned}
$$

We assume the PACTL formula $\mathcal{P}_{[0.3,0.4]}\Big(\mathtt{true}\,\mathcal{U}\,(\mathcal{X}_{msgFail})\mathtt{true}\Big)$ *remains the same for the analysis on* T_E*. The formula is a reachability analysis of the action "msgFail" on the root node*

($E_{root} = S$). The words reaching the "msgFail' from S are then recursively calculated for the change-affected (i.e., bold annotated terms in Figure 27) SRE terms respectively.

The probability functions of matching, prefix, suffix, and infix are calculated for every SRE term and propagated until all terms are covered. The results of the probabilistic functions already exist for the actions, and there is no need to calculate them. A new action "repair" is added to the model; therefore, its computation is newly realized:

$$[\![\mathtt{repair}]\!]_{infix,suffix,prefix}\mathtt{msgFail} = 0.0. \tag{105}$$

From the initial calculation provided in illustrative example in Chapter 4, we know that the only action that matches and includes "msgFail" is itself with probability 1; the probability of all other actions except "msgFail" to match 'msgFail" or have 'msgFail" as prefix, suffix, and infix are 0.

Similarly, the terms E_3, E_4, E_5, E_6, and E_7 are not re-computed for the probability functions, instead the pre-computations are reused and composed during the update of the tree e.g., the pre-computation of E_3 is present in E_2.

The newly added SRE terms E_8 and E_9 are calculated as follows.

$$\begin{aligned}[\![E_8]\!]_{infix}\mathtt{msgFail} = {} & \overbrace{[\![\mathtt{msgFail}]\!]_{suffix}\mathtt{msgFail}}^{1} \cdot \overbrace{[\![\mathtt{repair}]\!]_{prefix}\varepsilon}^{1} \\ & + \overbrace{[\![\mathtt{msgFail}]\!]_{suffix}\varepsilon}^{1} \cdot \overbrace{[\![\mathtt{repair}]\!]_{prefix}\mathtt{msgFail}}^{0} = 1\end{aligned} \tag{106}$$

Similar to the Kleene function 48 in Chapter 4, the search string for infix is a single action, but not an action sequence, which means we search the action "msgFail" in E_8. Therefore the iterations end in 1 step.

$$[\![E_9]\!]_{infix}\mathtt{msgFail} = 0.75 \cdot \overbrace{[\![E_8]\!]_{infix}\mathtt{msgFail}}^{1} = 0.75$$

E_2 needs to be updated with the a newly replaced E_8 as follows:

$$\begin{aligned}[\![E_2]\!]_{prefix}\mathtt{msgFail} = {} & \overbrace{[\![E_3]\!]_{prefix}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} \\ & + \overbrace{[\![E_9]\!]_{prefix}\mathtt{msgFail}}^{0.75} \cdot \frac{5}{100} = \mathbf{0.0375}\end{aligned} \tag{107}$$

$$\begin{aligned}[\![E_2]\!]_{suffix}\mathtt{msgFail} = {} & \overbrace{[\![E_3]\!]_{suffix}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} \\ & + \overbrace{[\![E_9]\!]_{suffix}\mathtt{msgFail}}^{0} \cdot \frac{5}{100} = \mathbf{0}\end{aligned} \tag{108}$$

$$\begin{aligned}[\![E_2]\!]_{infix}\mathtt{msgFail} = {} & \overbrace{[\![E_3]\!]_{infix}\mathtt{msgFail}}^{0} \cdot \frac{95}{100} \\ & + \overbrace{[\![E_9]\!]_{infix}\mathtt{msgFail}}^{0.75} \cdot \frac{5}{100} = \mathbf{0.0375}\end{aligned} \tag{109}$$

Using the calculations of E_2*, we update the results of* E_1 *as follows:*

$$\begin{aligned}[\![E_1]\!]_{suffix}\mathtt{msgFail} &= \overbrace{[\![\mathtt{sendMsg}]\!]\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]_{suffix}\varepsilon}^{1} \\ &+ \overbrace{[\![\mathtt{sendMsg}]\!]\varepsilon}^{1} \cdot \overbrace{[\![E_2]\!]_{suffix}\mathtt{msgFail}}^{0.0375} = 0.0375\end{aligned} \tag{110}$$

$$\begin{aligned}[\![E_1]\!]_{prefix}\mathtt{msgFail} &= \overbrace{[\![\mathtt{sendMsg}]\!]_{prefix}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]\varepsilon}^{1} \\ &+ \overbrace{[\![\mathtt{sendMsg}]\!]_{prefix}\varepsilon}^{1} \cdot \overbrace{[\![E_2]\!]\mathtt{msgFail}}^{0} = 0\end{aligned} \tag{111}$$

The calculation of the infix for the concatenation operation, the possibilities of the events that the decompositions of E_1 *can include the "msgFail" in different positions are also taken into account (C). As a result, the union of the probabilities* $A \cup B \cup C = A + B + C - (A \cdot B - B \cdot C - A \cdot C + A \cdot B \cdot C)$ *is calculated.*

$$\begin{aligned}[\![E_1]\!]_{infix}\mathtt{msgFail} &= \underbrace{\overbrace{[\![\mathtt{sendMsg}]\!]_{suffix}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![E_2]\!]_{prefix}\varepsilon}^{1}}_{A} \\ &+ \underbrace{\overbrace{[\![\mathtt{sendMsg}]\!]_{suffix}\varepsilon}^{1} \cdot \overbrace{[\![E_2]\!]_{prefix}\mathtt{msgFail}}^{0.0375}}_{B} \\ &+ \underbrace{\overbrace{[\![\mathtt{sendMsg}]\!]_{infix}\mathtt{msgFail}}^{0} + \overbrace{[\![E_2]\!]_{infix}\mathtt{msgFail}}^{0.0375}}_{C} \\ &- A \cdot B - B \cdot C - A \cdot C + A \cdot B \cdot C = 0.0375\end{aligned} \tag{112}$$

$$\begin{aligned}[\![S_2]\!]_{infix}\mathtt{msgFail} &= \overbrace{[\![E_1]\!]_{suffix}\mathtt{msgFail}}^{0.0375} \cdot \frac{65}{100} \\ &+ \overbrace{[\![E_5]\!]_{suffix}\mathtt{msgFail}}^{0} \cdot \frac{20}{100} \\ &+ \overbrace{[\![\mathtt{authenticationFail}]\!]_{suffix}\mathtt{msgFail}}^{1} \cdot \frac{15}{100} = 0.0375 \cdot 0.65 \approx \mathbf{0.024}\end{aligned} \tag{113}$$

The results for S_1 *can be easily obtained through "start" and "login" actions.*

The reachability of "msgFail" is then:

$$\boxed{\begin{aligned}[\![S]\!]_{infix}\mathtt{msgFail} &= \overbrace{[\![S_1]\!]_{suffix}\mathtt{msgFail}}^{0} \cdot \overbrace{[\![S_2]\!]_{prefix}\varepsilon}^{1} \\ &+ \overbrace{[\![S_1]\!]_{suffix}\varepsilon}^{1} \cdot \overbrace{[\![S_2]\!]_{prefix}\mathtt{msgFail}}^{0.024} = \mathbf{0.024}\end{aligned}} \tag{114}$$

The previous value of the reachability probability was 0.035*. Hence, we could obtain a lower probability of reaching "msgFail" by this change.*

At this point, SREs are locally calculated and do not require any traversal on the graph for reachability. Note that the idea lies upon calculating all information on every SRE term and composing solutions based on the operations.

The illustrative example of changing the SRE tree aims to demonstrate the localization of the model checking analysis over the SRE formalism. The parsed tree of the SRE model with changing sub-tree ($1 \leftarrow 2$) is presented in Figure 28. The tree is updated from E_4 up to E_3, E_2, E_1, and finally E_{root} in order.

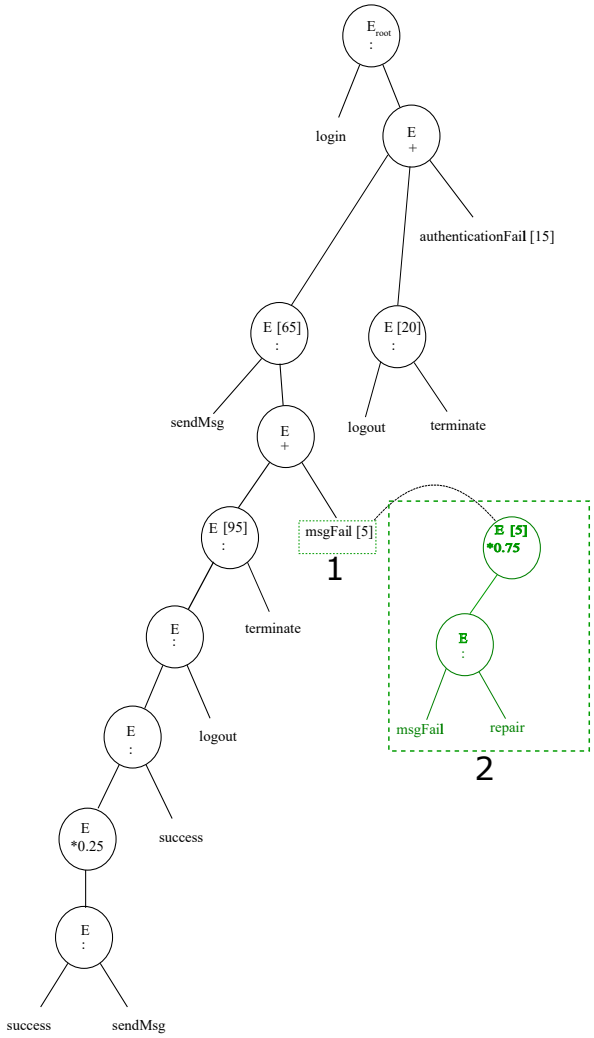


Figure 28: Changing SRE tree

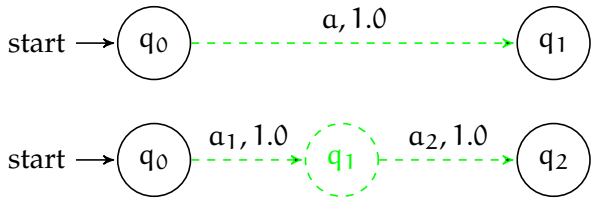
In the next subsection, we introduce some change patterns that support incremental verification using the above-mentioned edit operations.

6.2.2 *Supporting Incremental Verification: Change Patterns in Probabilistic Specification*

In this section, we designate possible change occurrences in a probabilistic model to ease the usability of *IQon* framework. These changes can be considered process models [74] or modifications for self-adaptive systems, such as service oriented models [145].

We describe all change types in terms of simple probabilistic Rabin automata (in practice, it is an explicit model represented as a transition matrix) and SRE tree operations.

Definition 6.5 (Addition of a Task)
It is an addition of a new task as an action between two events. In dynamic and adaptive systems, any task can bear at runtime. A task can be driven by a new action or a state.



The SRE Tree is manipulated for a task addition by adding a sub-term where the task sequence is added. Let E_T be a tree and $E = E_1 : E_2$ be a concat term with id i. The change is applied by addition of "a" (version 1) or addition of "a_1" and "a_2" sequentially (version 2) in between E_1 and E_2 concatenation sub-terms.

Version 1: We add two new *concat terms* $E_3 = a : E_2$ and $E = E_1 : E_3$. The tree is updated in the height of E. The change is represented by the following edit operations:
Precondition: Type of E is concat.
Input: E_{id},a.
SRE Edit operations: A new action a is added, and the corresponding concatenation term E is updated.

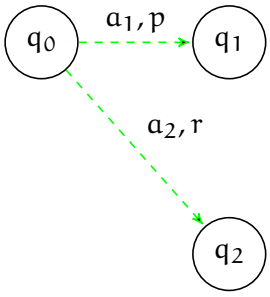
$$\Delta = \{[a, +], [E_3, a, 1, +], [E_3, E_2, 2, +], [E_1, E_2, 2, E_3, \rightleftarrows]\}$$

Version 2: We add two new *concat terms* $E_3 = a_2 : E_2$ and $E_4 = a_1 : E_3$. Then, $E = E_1 : E_4$. Then, the tree is updated in the height of E . The change is represented by the following edit operations:
Precondition: Type of E is concat.
Input: E_{id},a.
SRE Edit operations: A new action a is added, and the corresponding concatenation node E is updated.

$$\Delta = \{[a_1, +], [a_2, +], [E_3, a_2, 1, +], [E_3, E_2, 2, +], [E_4, a_1, 1, +], [E_4, E_3, 2, +][E_1, E_2, 2E_4, \rightleftarrows]\}$$

In an explicit model, the changes are applied by addition of one state and two transitions to the existing states. In a probability matrix, these changes will lead to the addition of one column and one row addition.

Definition 6.6 (Updating Distribution)
The probabilistic distribution of models tend to change based on the usage profile, especially at runtime environment in the long run of the software. The information can be obtained from running software depending on the frequency of events occurrence or the profile.

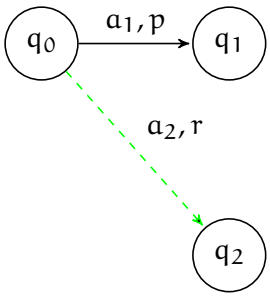


Precondition: Type of E is a sum node.
Input: E_{id}, id of sub-nodes, rates.
In terms of statistical data (distribution, profile, etc.), **the SRE model** includes more information instead of probabilistic distributions, such that the choice components stores the number, how many times the component is executed or called, instead of probabilistic discrete distributions. One can calculate the probability by diving into sum. Therefore, updating statistical data differ slightly for these models. SRE models gets the Δ as an update occurrence of the node i edit operation, and we change only one component. "Update-tree" finalizes the change propagation.
SRE Edit operations:

$$\Delta = \{[E_{id}, E, p, p', \circlearrowright\}$$

However, in a **PRA**, once a transition probability is changed, the other outgoing transition probabilities need to be updated for the corresponding state. In a probability matrix, such a change leads to the update of columns for one row.

Definition 6.7 (Changing the next action on the path)
In a dynamic and adaptive environment, any task or event can change its path on the way. For example, any update of "hazardous" paths, states, or cases might bear a new behavior on the next action through the path. In terms of modeling view, we discuss how such a change affects the structure of the model behaviour.



Precondition: Type of E is either *concatenation* or *choice*.
Input: E_{id}, sub-term to delete, the new term to add (i.e., replacement of the sub-term).
SRE Edit operations:

$$\Delta = \{[E_{id}, E, i, E', \rightleftarrows\}$$

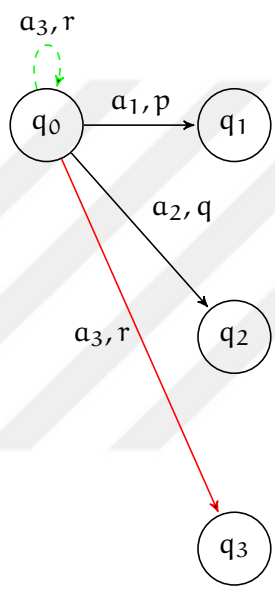
This change is restricted to one outgoing transition from q_0. More explicitly, a concatenation term is updated with the new term q_2. If q_2 does not exist, a new row should be added to the transition matrix.

Definition 6.8 (Changing an alternative path to a loop)
In this type of change, we investigate the case of removing an alternative path, and instead adding a self-loop. Two versions of this case is illustrated in the following figure.

Precondition: Type of E is a *choice* term.
Input: E_{id}, sub-term to change (e.g., E_3 that corresponds to (a_3, r) in the following figure).

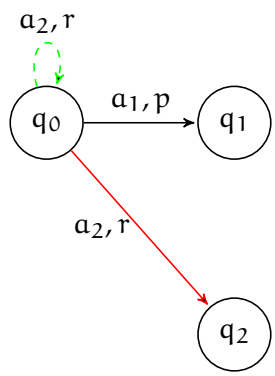
Version 1: In the first version, the sub-term E_3 is changed to a loop as a *Kleene* term, $E'_3 = a_3^{*(\frac{r}{r+p+q})}$. The choice term is updated $E = E_1 + E_2 + E'_3$ accordingly.
SRE Edit operations:

$$\Delta = \{[E'_3, a_3, \frac{r}{r+p+q}, +], [E_{id}, E, 3, E_3\prime, \rightleftarrows]\}$$



Version 2: In the latter version, one sub-term will be removed from the *choice* node. Therefore, the choice term will transform to a *concatenation* term whose first sub-term is a *Kleene* term. In other words, $E = E_{1[p]} + E_{2[r]}$ will transform to $E' = E'_1 : E_2$ and $E'_1 = a_2^{*(\frac{r}{r+p})}$.
SRE Edit operations:

$$\Delta = \{[E'_1, a_2, \frac{r}{r+p}, +], [E_1, E'_1, \leftrightarrows], [E', E_1\prime, 1, +], [E', E_2, 2, +], [E_{id}, E' \rightleftarrows]\}$$



In a transition matrix, this change will be applied as updating the corresponding cell.

Definition 6.9 (Changing a loop as a choice path)
A repetitive task can execute the next action as a choice. This case will be the reverse of the case provided in the definition above 6.8. Similarly, we present it in two sub-cases.

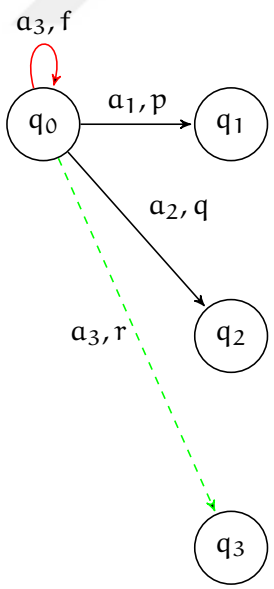
Precondition: Type of E is a *choice* node, at least one type of its children is *Kleene closure*.
Input: E_{id}, id of Kleene node.

Version 1 : The critical point of this change in an SRE is to represent the loop probability f of a_3 as a choice rate. The probability distribution of the choice paths of a_1 and a_2 corresponds to $1-f$ (the complement of the loop). Therefore, we distribute the $1-f$ to $\frac{p}{p+q}$ and $\frac{q}{p+q}$ by multiplying it for the paths a_1 and a_2, respectively. The normalized distribution from q_0 is then

$$\{(a_1, (1-f)\cdot\frac{p}{p+q}), (a_2, (1-f)\cdot\frac{q}{p+q}), (a_3, f)\}.$$

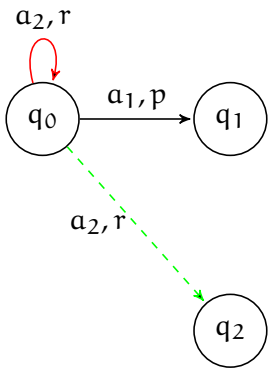
The rates of the sub-terms are obtained as integer values by multiplying their probabilities 10^k, $\exists k \in \mathcal{N}$.
SRE Edit operations:

$$\Delta = \left\{ \left[E_{id}, a_3, 3, f\cdot k, +\right], \left[E_{id}, 1, k\cdot(1-f)\cdot\frac{p}{p+q}), \circlearrowleft\right], \left[E_{id}, 2, k\cdot(1-f)\cdot\frac{q}{p+q}), \circlearrowleft\right] \right\}$$



Version 2 **SRE Edit operations:**

$$\Delta = \left\{[E_{id}, a_2, f\cdot n, +], [[E_{id}, a_1, 1, n\cdot(1-f)\cdot p], \circlearrowleft], [E', E_1\prime, 1, +], [E', E_2, 2, +], [E_{id}, E' \rightleftarrows]\right\}$$



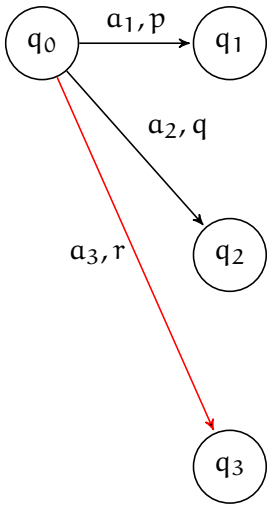
Definition 6.10 (Removing an alternation)
In the long run of software, events can execute alternative paths at runtime. Nevertheless, these alternatives might change during the evolution.

Precondition: Type of E is a *choice* node.
Input: E_{id}, node to delete.

Version 1: In this version, one can remove an alternation from multiple options more than 2, which will result updating an *SRE node* from $E = E_1 + E_2 + E_3$ to $E' = E_1 + E_2$ by removing node E_3.
SRE Edit operations:

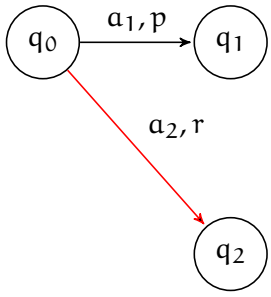
$$\Delta = \{[E_{id}, E, i, -]\}$$

In an explicit model, one removes the column and row that represent the state q_3 (removing the state) and update the cell of (q_0, q_3) as 0.



Version 2: If there are two alternative states and one is removed, such a change makes a different impact on the *SRE node*, such that we replace the node E with its sub-node that is left standalone (i.e., E_1). Node $E = E_1 + E_2$ is updated to $E = E_1$.
SRE Edit operations:

$$\Delta = \{[E_{id}, E, 2, -], [E_{id}, E, 1, \leftrightarrows]\}$$



Definition 6.11 (Adding a new alternation)
As opposed to the definition in 6.10, a new alternative paths can occur at runtime. It leads to increase the number of the choice elements in the next action. Similarly, we replace the node $E_1[r_1] + E_2[r_2]$ *with* $E' = E_1[r_1] + E_2[r_2] + E_3[r_3]$*, or the concat node* $E = E_1$ *will be replaced by a freshly defined choice node* $E' = E_1[r_1] + E_2[r_2]$ *in the case of the first occurrence of alternation.*

Precondition: Type of E is a *choice* node.
Input: E_{id}, node to add.
SRE Edit operations:

$$\Delta = \{[E_{id}, E, i, n, +]\}$$

Definition 6.12 (Removing repetition)
A repetitive action in the behavior might end a single action can be executed.

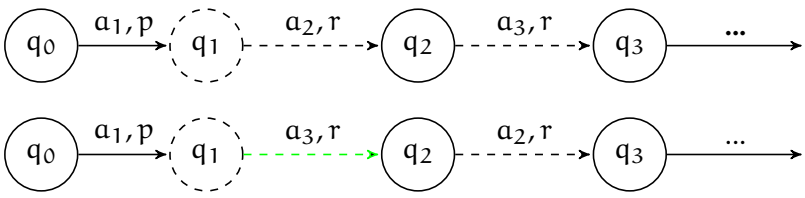
Precondition: Type of E is *Kleene* node.
Input: E_{id}, one event can round out its repetition and be executed once or zero.
SRE Edit operations:

$$\Delta = \{[E_{id}, f, -]\}$$

Definition 6.13 (Reordering sequential events)
During the operation of software processes, events can exchange their execution order in the course of time. We specify the change as "reorder action α *with* β*". Changing the sequence of the events affects the model in various magnitudes. In an SRE tree, the depth of the concat node(s) and in which order the sub-terms are concatenated are crucial. More precisely, a concatenation with two children* $E = E_1 : E_2$ *is the replacement of the node* $E' = E_2 : E_1$*.*

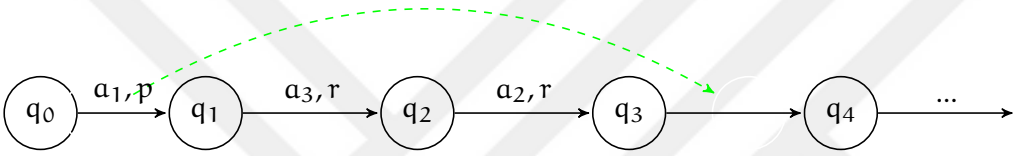
Precondition: Type of E is a *concat* node.
Input: E_{id}, nodes to reorder (i, j).

In case of multiple actions in the concatenated sequence, the SRE tree has multiple *concatenation* operations as well. The main idea is to divide the concatenation into two parts from 1 to i and from j to n, respectively, meaning that the original SRE term $E_1 : \ldots E_i : E_j \ldots : E_n$. The change occurs with swapping E_i with E_j. In practice, this change can cause to change the whole sub-tree.



Definition 6.14 (Moving the location of events)
During the operation of software processes, events can move an event to another place in the execution. We specify the change "move action" as (α,2) meaning that the action should be moved 2 step further if applicable.

Precondition: Type of E is a *concatenation* node, move step $n <$ number of children.
Input: E_{id}, node to move, n
Although the change specification is simple, the effect of this change might be costly. In the following figure, we move the action α_1 that is positioning after q_0 to two step further behind the q_3. In an SRE node, $E = a_1 : a_2 : a_3$ will be changed to $E = a_2 : a_3 : a_1$, which requires node deletion and addition operations. If there exist already an action in that position, meaning that $E = a_1 : a_2 : a_3 : a_4..$, such a case will lead to applying "move change ". Consecutively, $E = a_2 : a_3 : a_1 : a_4 \ldots$ as long as there are concatenated events. Such a change is not applicable if the next node is "Sum" or "Kleene" node.



The change patterns given above can be represented with a single or multiple edit operations. These patterns can be extended by domain knowledge or collected data as a result of monitoring of evolving systems.

Model check $(E, \Delta = \{(E_{id}, Op), .., (E_{id}, Op)\})$
Data: An SRE tree T_E, target action sequence s, Δ
Result: The reachability probability for a given action/action sequence s
{
foreach (E_{id}, Op) **do**
  search(E_{id})
(1) **if** Op *is a change pattern* p **then**
(2) apply (E_{id}, p)
  else
  apply (E_{id}, Op)
  end
end
return E
}
apply (E_{id}, op) {
T_E=contructSREnodewithValues(E_{id});
}
updateTree(T_E, E_{id})

Algorithm 7: Updated incremental model checking algorithm (Algorithm 6) with change pattern realization

The main goal of these patterns is to support the incremental model checking process in the *IQon* framework. An updated version of the incremental model checking is provided in Algorithm 7 (lines 1 and 2). Hence, applying any change contributes to

the user to define high level edit operations and can also increase the efficiency of the process.

6.3 PRELIMINARY EVALUATION: BENEFITS OF DOMAIN SPECIFIC EDIT OPERATIONS FOR CHANGING MODELS

This section provides a time performance comparison of our framework **IQon** with the SRE@SiDECAR framework. SiDECAR is a generic *incremental* verification framework for any language described in the operator precedence grammar [4] and the verification values over synthesized attributes [94].

6.3.1 *The SiDECAR Framework*

Bianculli et al. propose a framework called SiDECAR [16] that allows to define a variety of verification procedures over a grammar. The framework is composed of two elements: (1) a formal grammar to describe the artefact to be analyzed and (2) an attribute schema encoding the analysis algorithms. It uses a special class of grammars, called Operator Precedence Grammars (OPGs) and *synthesized* attributes.

SiDECAR follows a *syntactic-semantic* approach. Put another way, it assumes the artefacts to be analyzed to have a syntactic structure compliant with a formal grammar, so that the analysis procedure is encoded as the computation of semantic attributes associated with the production rules of the formal grammar. The generic framework gets a formal grammar described in the OPG and builds the abstract syntax tree (AST) [70] of the artefact and then, computes the semantic attribute resulting from the analysis of that portion for each node of the AST [16].

Whenever a portion of the artefact description changes, SiDECAR identifies the boundaries of the change using the locality property of OPGs and recompute the corresponding semantic attributes. The computation will be propagated along the AST up to the root, merging newly computed partial results with those previously computed for the unchanged part of the artefact description.

We demonstrate an example from [16] to give a glimpse of how the incremental evaluation of an arithmetic expression would be performed in SiDECAR.

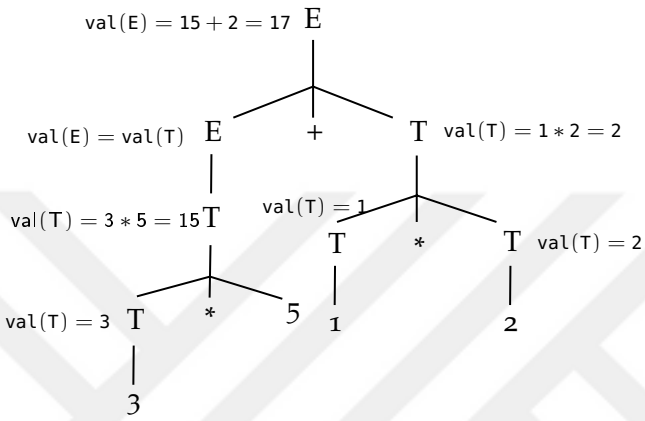
The formal grammar to represent arithmetic expression is reported in Figure 29a, alongside its attribute schema. The grammar is an OPG and allows to describe all the arithmetic expressions composed of sums of products of numbers. The terminal symbol '**n**' represents any numerical value, while '**+**' and '*****' stand for the addition and multiplication operators, respectively. The non-terminal symbol S represents the axiom of the grammar, i.e., the starting point for the generation of any arithmetic expression. The right of each syntactic production rule corresponds to an attribute synthesis procedure (Figure 29b). Such a procedure is formalized through the function $val(\cdot)$. The function allows to compute the numerical value of the left-hand-side symbol as a function of the values of the symbols appearing at the right-hand side of the production rules. The helper function $val(\cdot)$ is used to compute the numerical value of a number from its syntactic representation (e.g., the string "3" will correspond to the integer number 3). The abstract syntax tree (AST) with attached attributes of arithmetic operations is provided in Figure 29c.

$S = E$
$S = T$
$E = E + T \mid T$
$T = T * n \mid n$

(a) OPG of arithmetic operations

$val(S) = val(E)$
$val(S) = val(T)$
$val(E) = val(E) + val(T)$
$val(E) = val(T) + val(T)$
$val(T) = val(T) * n$
$val(T) = n$

(b) Attribute grammar of arithmetic operations



(c) AST of arithmetic operations

Figure 29: Plugin components of arithmetic operations in SiDECAR [16]

Let us introduce the SRE formalism as a SiDECAR plugin in the following subsection.

6.3.2 *The SRE@SiDECAR Plugin*

We have created an operator precedence grammar of an SRE as the first component of the SiDECAR framework. The syntax of an SRE (equation 104) is represented as an OPG in Figure 30a that is created based on the SRE syntax, albeit with a simplified version (e.g., removing syntactic sugars such as *plus closure*, *guarded choice*). The parentheses encapsulating an SRE term E remain in the grammar since it preserves the separation of the terms, e.g., $(a + b) + c$ and $a + (b + c)$ during the translation of automata The SiDECAR framework automatically detects if the provided input grammar is an OPG and produces the operator precedence table as presented in Figure 30b. The operator precedence table presents the operation precedence relations, i.e., $>, <, =$, and the symbol # represents the end-of-line.

$$
\begin{aligned}
S &= A \mid B \\
A &= A[n] + B[n] \mid B[n] + B[n] \\
B &= B : B \mid B * d \mid a * d \mid (B) \mid (A) \mid a
\end{aligned}
$$

(a) Operator precedence grammar (OPG) of SREs

	*	)	p	+	r	#	a	(	[	]	:
*			=								
)	>	>				>			>		>
p	>	>				>			>		>
+	<					<	<	=			<
r										=	
#	<						<	<	<		<
a	>	>				>			>		>
(	<	=					<	<	<		<
[					=						
]		>		=		>				>	
:	>	>				>	<	<	>		>

(b) Operator precedence table (OPG) of SREs

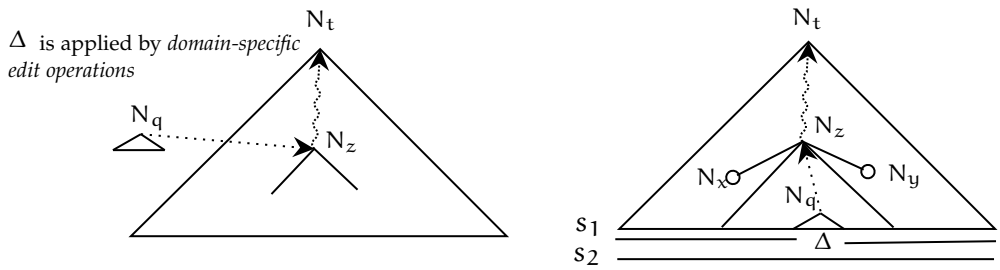
Figure 30: An OPG of SREs as a SiDECAR component

Another input required for the SiDECAR framework is an attribute schema that can be seen in Table 5 for the SRE syntax. The helper function $val()$ stores the probabilistic values of the SRE term that are calculated according to the production rules from its syntactic representation. Specifically, these functions are probabilistic *matching*, *prefix*, *suffix*, and *infix* during the reachability analysis, whose mathematical computations are listed in Chapter 4. The reachability analysis is then performed using these computations on the top of SiDECAR.

$$
\begin{aligned}
val(S) &= val(A) \\
val(S) &= val(B) \\
val(A) &= val(A)[n] + val(B)[m] \\
val(A) &= val(B)[n] + val(B)[m] \\
val(B) &= val(B) : val(B) \\
val(B) &= val(B)^{*}f \\
val(B) &= val(a)^{*}f \\
val(B) &= (val(B)) \\
val(B) &= (val(A)) \\
val(B) &= val(a)
\end{aligned}
$$

Table 5: An attribute schema of the SRE plugin on SiDECAR

We compare *IQon* with the SRE formalism on the SiDECAR framework to discuss the benefits and drawback of our approach. As described above, SiDECAR is a generic method that requires an OPG syntax and an attribute schema. The syntax of the SRE and the formal model checking algorithm framework can be built up smoothly due to compositional probabilistic functions. However, the main challenge occurs during the change input and finding the affected part. In SiDECAR, a change is introduced in the given syntax. A visual representation of the difference between the *IQon* and the SiDECAR principle is provided as an overview in Figure 31.



(a) *IQon* principle **(b)** SiDECAR methodology [16]

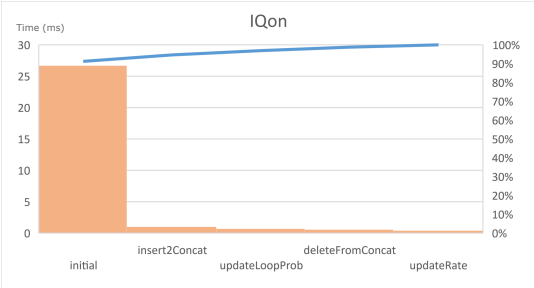
Figure 31: *IQon*'s working principle versus Generic SiDECAR methodology

Being taken from the previous SRE model versions (initial and changed models in Figure 27), Example 6.2 demonstrates a string difference between two SRE models for the SRE@SiDECAR framework. The SiDECAR gets the difference between strings and searches the smallest non-terminal that encapsulates the difference. Such a search technique can be inefficient when the change is represented as a long string. More explicitly, a change inside parentheses will result in the whole string counted as a Δ. The consequences of such cases appear in the following experiments. Identifying and optimizing a change in the string that obeys the SRE grammar (e.g., in $a : b$, removing a will conclude in an incorrect grammar instance and not applicable for running the change action) might be cumbersome for the user. Hence, we provide domain specific edit operations that are familiar to the user in probabilistic models.

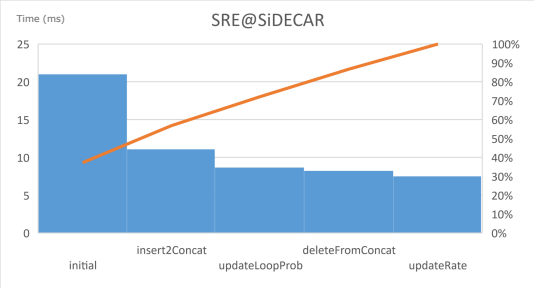
Example 6.2 (Representation of Δ in example given in Figure 27 for the SiDECAR plugin)

$$\begin{aligned}
\textit{SRE model: } x_0 &= \mathtt{login} : (\mathtt{sendMsg} : (((\mathtt{success} : \mathtt{sendMsg})^{*0.25} \\
&: \mathtt{success} : \mathtt{logout} : \mathtt{terminate})_{[95]} + \mathtt{msgFail}_{[5]})_{[65]} \\
&+ \mathtt{logout} : \mathtt{terminate}_{[20]} + \mathtt{authenticationFail}_{[15]}) \\
\textit{changed SRE model: } x_0 &= \mathtt{login} : (\mathtt{sendMsg} : (((\mathtt{success} : \mathtt{sendMsg})^{*0.25} \\
&: \mathtt{success} : \mathtt{logout} : \mathtt{terminate})_{[95]} + (\mathtt{msgFail} : \mathtt{repair})^{*0.75}_{[5]})_{[65]} \\
&+ \mathtt{logout} : \mathtt{terminate}_{[20]} + \mathtt{authenticationFail}_{[15]}) \\
&\textit{The } \Delta \textit{ is calculated using string difference.} \\
&\Delta = (\mathtt{msgFail} : \mathtt{repair})^{*0.75}
\end{aligned} \tag{115}$$

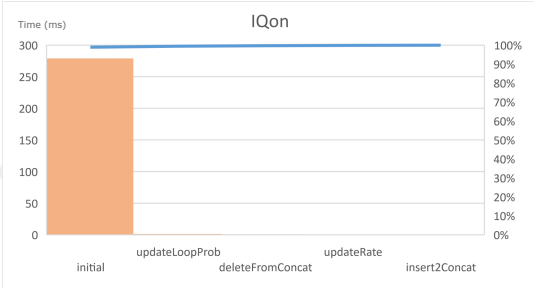
The time comparison is held on various SRE sizes from 400 up to 300000, as reported in Figure 32. The average execution time measurements (as millisecond) are clustered for different edit operations and presented for both *IQon* and SRE@SiDECAR frameworks for each data set. In each line (Figure 32a- Figure32b, Figure 32c- Figure32d, Figure 32e- Figure32f, Figure 32g- Figure32h), same input is evaluated in terms of time measurements for IQon and SRE@SiDECAR respectively. In general, the SRE@SiDECAR exploits higher performance comparing to *IQon* for initial model constructions, e.g., *IQon* takes longer time especially for the data sets represented in Figure 32e and in Figure 32g (the real values are 3 times bigger that cannot be demonstrated in the graph to show the nuances for the time elapsed during the edit operations). However, *IQon* outperforms it by far for the changing models using the domain-specific edit operations.



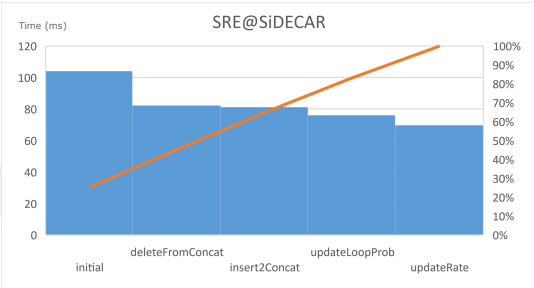
(a) *IQon* execution time (ms), SRE length between 100-500



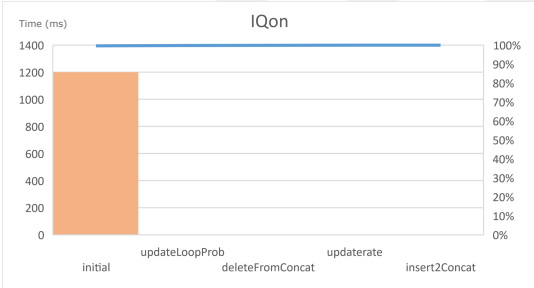
(b) SRE@SiDECAR execution time (ms), SRE length between 100-500



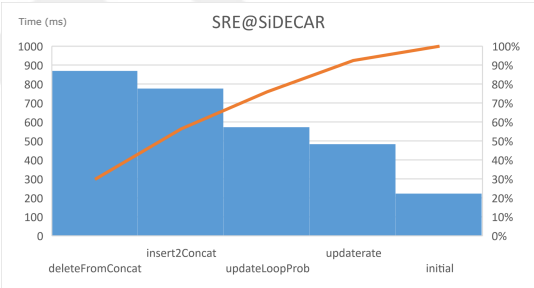
(c) *IQon* execution time (ms), SRE length between 1000-9000



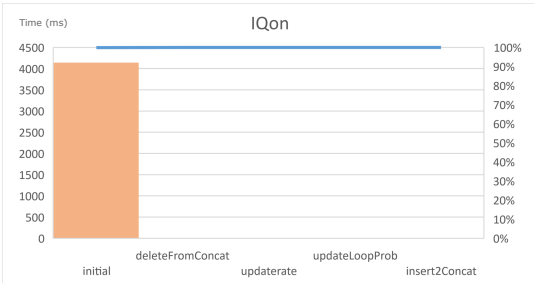
(d) SRE@SiDECAR execution time (ms), SRE length between 1000-9000



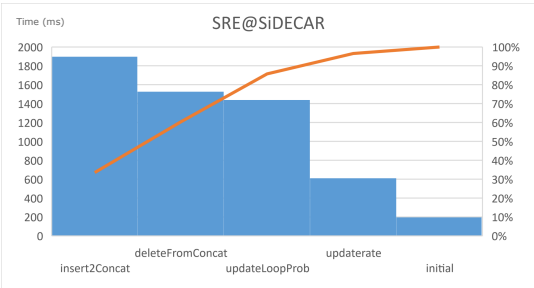
(e) *IQon* execution time (ms), SRE length between 10000-90000



(f) SRE@SiDECAR execution time (ms), SRE length between 10000-90000



(g) *IQon* execution time (ms), SRE length between 100000-500000



(h) SRE@SiDECAR execution time (ms), SRE length between 100000-500000

Figure 32: Comparison of *IQon* and SRE@SiDECAR for the same changes

6.4 INCREMENTAL TRANSFORMATIONS BETWEEN PRA AND SRE MODELS

Equivalence between probabilistic automata and stochastic regular expressions has been shown in the previous chapters (Chapter 5). Hence, transformations between a PRA and a SRE are definable. However, if a change occurs in a PRA or in an SRE, it might not be a *one to one* mapping once the change is propagated to the other model. In this section, we will discuss the *change* effect between models for the *change* propagation. Incremental transformations might avoid re-transformations of the full models and help to identify relations between the PRA and SRE.

6.4.1 *Incremental Transformations from PRA to SREs*

We assume that a change, which is represented by a Δ, has been applied to a PRA. If the change is reasonably small, it might be more efficient to apply the delta to the transformation that has been already performed instead of transforming the model from the scratch again.

Ideally, atomic changes consist of adding or removing a transition or a state. In 5, we have described how to solve equations for an SRE for a given PRA. During the calculation, the coefficients in the equations and their probabilities can be stored as references. After the initial transformation, once we have a change in the PRA, we replace the references by the actual values for the concrete probabilistic automata. Let us explain the approach with the following example (Figure 33).

Example 6.3 (Storing the SRE equations during the translation from the PRA)

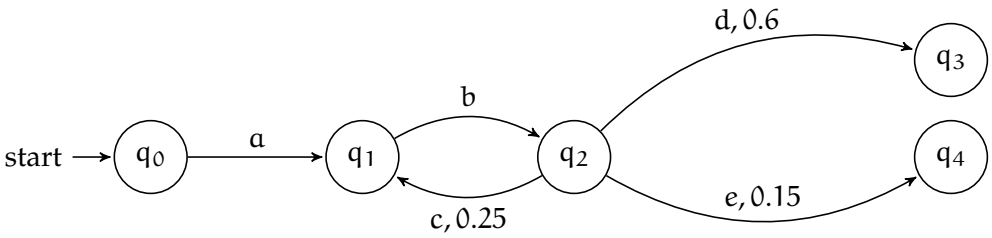


Figure 33: Example probabilistic Rabin automata with equations

$$
\begin{aligned}
x_0 = a : x_1 \Rightarrow & \qquad a : x_1 \\
x_1 = b : x_2 \Rightarrow & \qquad (b : c)^{*0.25} : b : (d_{[60]} + e_{[15]}) \\
x_2 = c : x_1 + d : x_3 + e : x_4 \Rightarrow & \qquad c : x_{1\,[25]} + d_{[50]} + e_{[15]})
\end{aligned}
$$

Once we solve the equation, the regular expression will be
$x_0 = a : (b : c)^{*0.25} : b : (d_{[60]} + e_{[15]})$ by Arden's lemma (see chapter 5). Once every expression is loop-free for every equation, we keep the coefficients, and transitions as references to the expression. For example, if any change occurs in the probabilistic

automata as removing "c" from the matrix or from the PRA as an atomic action, we can update all the equations that include "x_1" and the corresponding expression.

This approach can be beneficial for incremental transformations by sacrificing space $O(n^3)$. In the worst case, all expressions are updated, that is, the whole model is transformed for all of the equation systems. The more sparse is the PRA, the more incremental the approach becomes.

The use of dynamic programming can speed up this process. We can reuse the equations we have already solved up to the point where a change actually happens to the model. It would provide at least some performance improvement, although the gain strongly depends on the order of the states , i.e., the index of the changed states.

A drawback of this solution is the final SRE can grow fast and could be large. To avoid such a growth, we apply simplifications on the fly during the transformations. These simplifications are listed as follows:

- Merging of nested concatenations: $a : (b : c) \Rightarrow a : b : c$
- Removal of ϵ in concatenations: $a : \epsilon : b \Rightarrow a : b$
- Merging of identical SREs in choices: $a_{[1]} + b_{[2]} + a_{[3]} \Rightarrow a_{[4]} + b_{[2]}$
- Merging of nested Kleene expressions: $(a^{*p})^{*q} \Rightarrow a^{1-\frac{1-q}{1-pq}}$
- Removal of Kleene expressions over ϵ: $\epsilon^{*p} \Rightarrow \epsilon$
- Removal of Kleene expressions with repetition probability zero: $a^{*0} \Rightarrow \epsilon$

6.4.2 *Incremental Transformations from SREs to PRA*

To apply incremental transformations from regular expressions to automata, we assume that the initial model of probabilistic automata is transformed from the corresponding SRE model by the construction method described in Section 5. Therefore, the model is not minimized. Such a sparse model enables to store every SRE node and its representing probabilistic automata. The change in the SRE is expressed as a set of substitutions of certain sub-trees. For instance, let $E = a : (b_{[1]} + c_{[2]}) : (d^{*0.3})$ be an SRE and the goal is to replace the d with $e : f$ then, a change Δ is:

$$\Delta = \{\texttt{\#ref(d)} \rightarrow e : f\}$$

The application of this delta results in $E' = a : (b_{[1]} + c_{[2]}) : ((e : f)^{*0.3})$.

The SRE node on which Δ is applied can be split into sub-nodes to avoid large replacements. Let us assume a change that swaps the last two concatenation components of E' which are $b_{[1]} + c_{[2]}$ and $d^{*0.3}$. The Δ would be applied as follows:

$$\Delta = \{\texttt{\#ref((b[1]+c[2]):(d*0.3))} \rightarrow d * 0.3 : (b[1] + c[2])\}$$

By allowing to reference sub-trees in the replacement, the delta can be split into

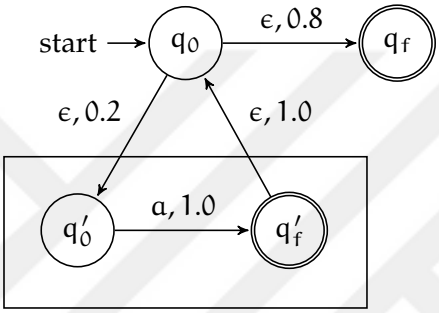
$$\Delta = \{\texttt{\#ref((b[1]+c[2]):(d*0.3))} \rightarrow \texttt{\#ref(d*0.3)} : \texttt{\#ref(b[1] + c[1])}\}$$

Note that this operation is constant.

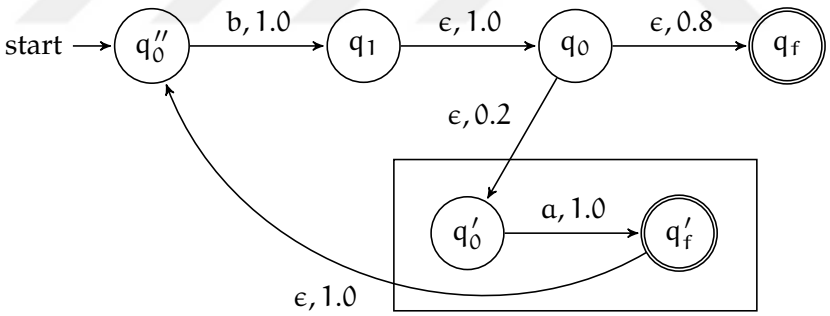
Once we apply the Δ, each construct of the SRE is stored as a references called *sub-PRA*. These references include a pointer to the initial state and to the final state. A sub-PRA is not simplified during the transformation once they are constructed. Therefore, a local change can be easily transformed by replacing the sub-PRA. The final PRA might be very large as a drawback. Let $M = (\Sigma, Q, q_0, P, F)$ be a PRA, and q_0, q_f be the initial and final states, respectively. (without loss of generality, we assume one final state for the sake of clarity). The algorithm that decomposes PRA into sub-PRA is applied as follows:

$$\Delta = \{\texttt{\#ref(a*0.2)} \rightarrow \texttt{b} : \texttt{\#ref(a*0.2)}\}$$

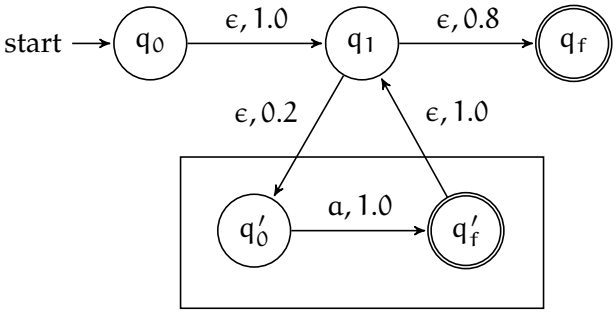
Applied transformation on the PRA results in:



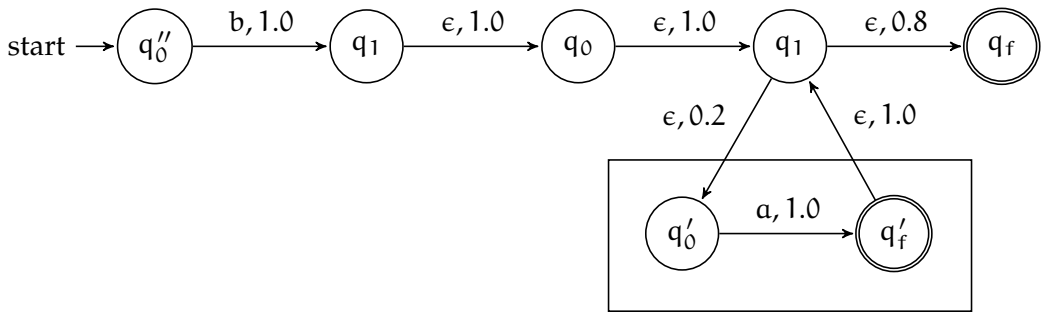
would become



while



correctly transforms to



6.5 CONCLUSION

In this chapter, we have presented our incremental quantitative verification framework *IQon* on the top of SRE model checking (Chapter 4) and its modularity for probabilistic computation. One of the main contribution of this thesis is to achieve the incremental quantitative verification by applying edit operations and change patterns on the SRE model checking formalism.

We discuss the applicability of our SRE model checking algorithm on the generic and efficient incremental verification framework called SiDECAR. Our SRE formalism is provided as an input for the SiDECAR framework, and any change is given in the SRE syntactically. We demonstrate how *IQon* framework outperforms the domain-independent framework SRE@SiDECAR plugin by applying the changes on SRE trees directly. We note that the SiDECAR uses a performance efficient operator precedence parsing technique and applicable for any syntax represented in an OPG. However, we use the benefit of domain specific change operations for the SRE checker. In the following chapter 7, we compare *IQon* with external probabilistic model checkers in the case of evolution.

Furthermore, we investigate possible incremental transformations between PRA and SREs and vice versa in Section 6.4. The observation is that these transformations are not one-to-one even if a mapping between model elements is captured.

Part IV

VALIDATION

7 EVALUATION

This section presents an evaluation framework to validate the provided formalism and the proposed approach in this thesis for evolving software. The evaluation framework constitutes of a couple of steps and an experimental setup for the validation of the incremental quantitative verification (*IQon*). Hence, one of the main research questions is "How applicable and efficient is the proposed approach?" To search for answers to this question, one should work on various data sets of SRE models whose data benchmarks are not available to our knowledge.

Therefore, we propose two methods *to obtain SRE models* as a *requirement/precondition* for the validation of the SRE model checking framework (an overview of the experimental setup is provided in Figure 34):

- A new approach that extends a regular expression learning method called BLA (Block-wise left alignment) for learning stochastic regular expressions. Such a new approach is validated by checking the conformance between the trace data and the obtained model.
- Application of transformations that is formally founded in Chapter 5.

In the sequel, we compare the verification results of SRE with PRISM probabilistic model checker to corroborate the formally founded SRE verification framework provided in Chapter 4. Finally, the validation of *IQon* is proceeded following the internal and external evaluations.

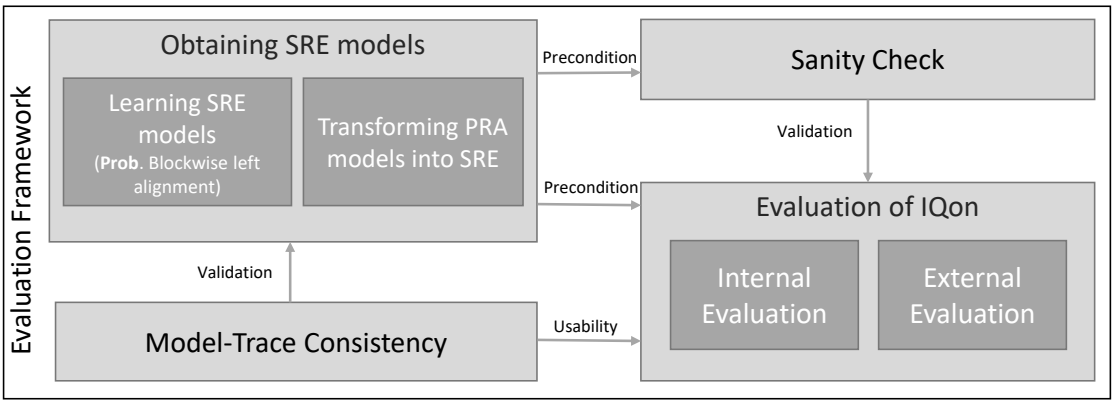


Figure 34: Experimental setup

7.1 OBTAINING SRE MODELS

In this section, we present the learning technique of SREs and transformations from PRA to SREs to obtain our data sets for the evaluation of the probabilistic model checking.

7.1.1 *Learning SRE Models*

There exist several approaches to learn regular languages in the literature e.g., grammar inference, model learning and grammar mining [132].

We have adapted a time-efficient learning method for the regular expressions in the probabilistic setting to learn the probabilistic characteristics of the sample set for the SRE inference. The original method, called Block-wise Left Alignment (BLA), was first introduced in the study of Fernau et al. [47]. The article describes an approach to learn simple (unambiguous) regular expressions from positive data (sample sets in our case). It also specifies the class of regular expressions that can be learned by the method.

Block-wise Left Alignment (BLA) Method

The method consists of three basic steps:(1) Block-wise Grouping, (2) Left-Alignment, and (3) Inference. We execute all steps over an example [47] provided as below. Consider the following example sample set, where $a, b \in \Sigma$ are the symbols of an alphabet Σ:

a	b	a	b	b
a	a	b	b	
a	b	a	b	a
a	b	c		

Table 6: Simple sample set on the alphabet $\Sigma = \{a, b, c\}$

1. *Blockwise Grouping:* Each trace from the sample set is transformed into a sequence of block letters. Any $[x^n]$ is called a block letter whenever x is a character in the alphabet over which the words are formed; and where $n \in \mathbb{N}^+$. Practically, we group any sequential repetition of a single character into the corresponding block letter (e.g., "a a" becomes $[a^2]$ or $[aa]$) (Table 7).

[a]	[b]	[a]	$\overbrace{[bb]}^{[b^2]}$	
[aa]	[bb]			
[a]	[b]	[a]	[b]	[a]
[a]	[b]	[c]		

Table 7: Block-wise grouped sample set on the alphabet $\Sigma = \{a, b, c\}$

2. *Left-Alignment:* The block-letter traces are aligned into columns, where ε denotes the empty word. If a left-aligned trace does not populate a full row, the remaining columns are filled with the empty word.

3. *Inferring a regular expression from the table:* In this step, a regular expression is constructed by assuming each column of the table to be a sub-expression of a concatenation. In our example, it yields the following expression:

$$(a|aa) : (b|bb) : (a|c|\varepsilon) : (bb|b|\varepsilon) : (a|\varepsilon)$$

[a]	[b]	[a]	[bb]	ε
[aa]	[bb]	ε	ε	ε
[a]	[b]	[a]	[b]	[a]
[a]	[b]	[c]	ε	ε

Table 8: Alignment of the grouped elements

More sophisticated version of inferring the regular expression which results in the following expression:

$$(a|aa) : (b|bb) : (\varepsilon|a(b|bb)(\varepsilon|a)|c)$$

This expression is more specific in a way that it does not allow the word "a b c b b", whereas the regular expression $(a|aa) : (b|bb) : (a|c|\varepsilon) : (bb|b|\varepsilon) : (a|\varepsilon)$ does. In our adoption of the algorithm, this extra-separation of branches is not implemented towards the end of the traces. To further generalize our results, "one or two repetitions" are considered a (very) special case of "one or more repetitions". In other words, we generalize block-letter alternatives in the regular expression of the same character to a plus-closure expression e.g., $(a|aa|c)$ to $(a^+|c)$. In our first regular expression, it yields:

$$a^+ : b^+ : (a|c|\varepsilon) : (b^+|\varepsilon) : (a|\varepsilon)$$

Probabilistic Block-wise Left Alignment Method (pBLA)

SREs and PRA are equally expressive and can be transformed into each other (See Chapter 5). However, one might question the use of an independent SRE learning method. Learning an automaton and transforming it to an equivalent SRE would be the first solution but not satisfactory for two main reasons:

Efficiency: For a user who would like to get an SRE as an input, the time to learn the model would be PRA learning time ① plus the time for the transformation from the PRA to the SRE. As we see later on in the performance evaluation of pBLA, a direct learning of an SRE is almost certain to be much faster than learning a PRA.

Model Quality: Although there is no universal measure for SRE model quality, the PRA-to-SRE transformation is designed to produce correct result but the size of the regex can explode up to $n^{\emptyset(\log n)}$, where n is the number of states [69]. However, a separate learning method for SREs can be designed and optimized for model checking.

Thus, the three-step functionality of Blockwise Left Aligned (BLA) guarantees a linear time complexity regarding the size of the sample set. Each trace in the sample set has to be handled a constant number of times as the steps of the algorithm specify. Since each character of a trace has to be examined, it is also expected to behave linearly with the sum of trace lengths of the input sample set.

We consider the probabilistic nature of our sample set to infer SREs using BLA method. More specifically, we simply introduce SRE rates to accommodate the probabilistic data in the first inference of a regular expression. It means that we add the number of traces, which have that specific block-letter in the current column to each alternative in the sub-expressions. The following example demonstrates the method.

① AAlergia is one of the well-known approaches to learn PRA [111].

Example 7.1 (Simple inference for SREs)
By following the same trace set in Table 6, $(a|aa)$ *is inferred in the first column. Looking at the table from step 2, 3 traces with a single occurrence of character* a *and one trace with two occurrences of the character* a *are obtained. Thus, we introduce the rates 3 and 1 accordingly:* $(a[3]|aa[1])$*. While further generalizing the expression, we also need to consider the repetition rates of the plus-closure, given that we have some alternatives* $[a^{n_1}]\dots[a^{n_k}]$*. The resulting generalized expression in an SRE is of the form* a^{+f}*, where* f *is the repetition probability of the expression* a*. Thus, we have to infer this probability from the given alternatives.*

Let $\#[a^n]$ (a being a character in the sample set alphabet and $n \in \mathbb{N}$) be a function that maps block-letters to the number of their occurrence in the current column. For example, $\#([a^1]) = 3, \#([a^2]) = 1$ in the first column of Table 8. We derive the loop probability f for a fixed column with the following formula:

$$f = 1 - \frac{\#([a^{n_{min}}])}{\sum_{n \in \{n_1 \dots n_k\}} \#([a^n])}, \qquad n_{min} = min(n_1 \dots n_k) \tag{116}$$

That is, the probability of repeating the character a is the proportion of the number of occurrences of the minimum number of repetitions $\#([a^{n_{min}}])$ and all other traces with this block-letter in the same column. We also impose a lower-bound condition on the structure of the SRE to achieve a more precise expression. For instance, if we observe block-letters $aaa, aa, aaaaa$ in a single column, the resulting sub-expression is $(a : a : a^{*f})$ rather than $(a : a^{*f})$. Otherwise, the proportions between $\#(aa)$ and $\#(aa) + \#(aaa) + \#(aaaa)$ would not be valid. In this example we compute $\alpha = 1 - \frac{1}{3} \approx 0.66$, thus the resulting SRE is $(a : a : a * (0.66))$. As a result, we have implemented a *probabilistic BLA learning method* called *pBLA* that allows us to infer SREs from an input sample set with the help of relevant modifications.

7.1.2 *Performance Evaluation of Learning SREs*

In this sub-chapter, we discuss the complexity of the proposed learning approach *pBLA* and compare it with the well-known AAlergia [111] learning algorithm in terms of time performance.

AAlergia is one of the approaches that learns a probabilistic automation model from system traces for *probabilistic model checking*. It basically introduces an algorithm using a method of merging nodes that integrate the sample set (in the form of a prefix tree) into a probabilistic deterministic finite automaton.

In the context of this algorithm, states are said to be equivalent if the outgoing transition probabilities for every symbol of the alphabet and the destination nodes are the same. Due to the statistical fluctuation, the equivalence is accepted within a confidence range, called *compatibility*. The algorithm uses the Hoeffding-Bound [41] to decide on the state compatibility. If this check, which is done for the termination probabilities and the outgoing transition probabilities of two nodes, succeeds, all destination nodes have to be checked recursively. After finding two compatible nodes in the graph, AAlergia merges all incoming and outgoing transitions including their frequencies as well as the termination frequencies. Thus, all the destination nodes are merged exhaustively to outcome deterministic models.

We have generated random sample sets of increasing size to evaluate the algorithms in terms of scalability regarding the input sample set. We have produced increasingly

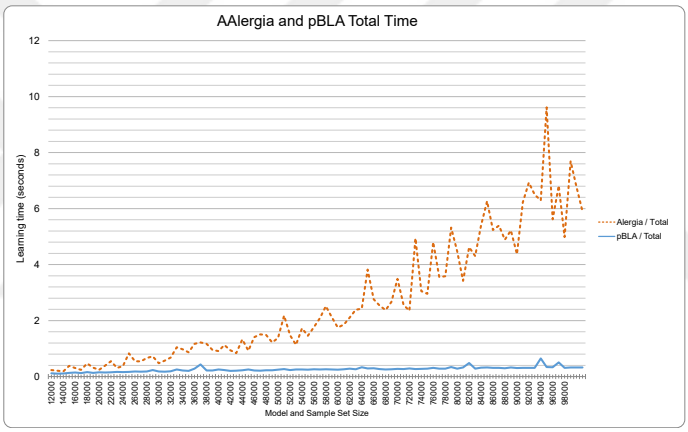
large sample sets from increasingly complex models. The measure of *model size* yields the parameters provided in Table 9 for the *Random Generator*.

Model size: M
Output Ranges:

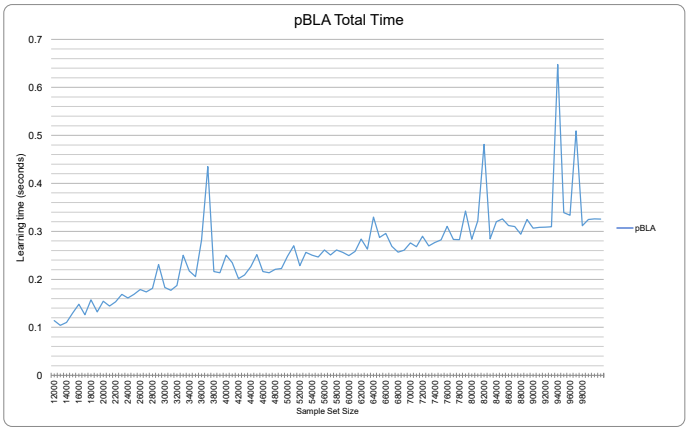
Number of states	max(2, M - 2) ... M + 2
Number of Outgoing Transitions per State:	max(4, 0.2 * M - 2) . . . max(5, 0.2 * M + 2)
Alphabet Size:	max(4, 0.28 * M - 2) . . . max(4, 0.28 * M + 2)
Self-cycle Probability:	0.1

Table 9: Model parameters

Our Performance Evaluation (see Figure 35) reports that pBLA runs in linear time compared to AAlergia's quadratic time complexity. However, due to the nature of the generalization technique used in the pBLA learning, it is not parametrizable as AAlergia is, whereas AAlergia provides the parameter that specifies a threshold for the computation of probabilistic similarity. As a drawback of the pBLA approach, it provides no such configuration. In future work, the pBLA learning algorithm might be evolved by adopting branch handling as described in [47].



(a) Execution time comparison between AAlergia and pBLA



(b) Internal execution time evaluation of pBLA

Figure 35: Performance evaluation in time for the pBLA learning method

In addition to the performance evaluation, we provide a validation of our results to see how consistent the learned language with the trace set is. We use a widely used technique in process driven engineering, called *conformance checking*, to measure the

conformance between the process model and the event log [3]. Explicitly, we adapt the question "Do the model and the log conform to each other?" from process engineering [3] to "Do the learned model and the software traces conform each other?" One of these techniques is footprint matrix method [129], where we study an extension of the algorithm for the probabilistic models. This algorithm and the tool is called ***ConsistAnts***, as an outcome of that investigation, are explained in detail in the following section.

7.1.3 *Consistency Checking of Probabilistic Models and Software (The ConsistAnts Framework)*

We have implemented a consistency checker that extends a conformance testing algorithm to evaluate what percentage the learned model is consistent with the trace set, and demonstrate how SRE modeling can be employed for software behaviour. Moreover, a generalized framework for "consistency checking of probabilistic models and software" is presented within the following problem and motivation.

> *Quantitative (software quality) models can be used for diverse cases in the development of large-scale software systems. For instance, developers can use such models to identify performance in distributed infrastructure as well as potential points of a failure. Accordingly, selective measures can be taken in order to improve the reliability, performance, or availability of systems (e.g., increased testing efforts or replications for heavily-used components) [9]. Developers may use learning algorithms [6], design them manually, or use a combination of both in order to obtain software quality models of a system. As a consequence, it must be ensured that the model accurately represents the system behavior, whenever the system evolves. Thus, consistency checking tools are required that can be used to confirm the conformance of the system behavior with its model and vice-versa.*

We study that problem and develop a generic tool called **ConsistAnts** that can get a trace set as an input, and produce consistency reports. The tool is fostered with features to identify counterexamples and detailed documentation of model-trace consistency.

In the context of the *ConsistAnts* tool, we have developed a novel framework that supports the consistency checking between learned/designed probabilistic models and software traces. The *ConsistAnts* framework, which is the first tool to our knowledge in this context, is implemented in terms of an Eclipse Plugin, but all its functionality is also exposed as a command line interface.

The term *consistency checking* is used as the term *conformance checking*, which is a concept in process mining that is used to ensure that event executions comply with the process model [3]. Conformance checking can also be used as a validation technique in model checking together with specifications [66] and preconditions [80]. Similar to [3], we propose a validation technique for software traces and probabilistic models. Hence, we have released (publicly available[②]) an extensible validation tool for learning algorithms and manually designed models in the assumption of existing software traces. The *ConsistAnts* tool adopts three main features:

② **https://github.com/ConsistAnts/ConsistAnts**.

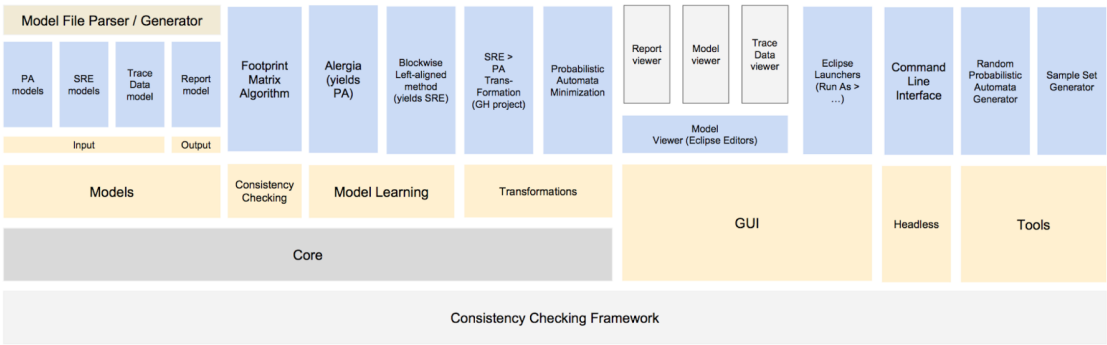


Figure 36: Architecture of the *ConsistAnts* Tool

1. Yielding a degree of consistency per trace length, a method for conformance checking of PRA and traces,
2. Taking software traces as input, an extensible validation platform for probabilistic learning algorithms,
3. A user-friendly visualization of the models as well as the conversions between them (Formally described in Chapter 5).

7.1.3.1 *The ConsistAnts Architecture*

The core architecture demonstrated in Figure 36 relies on the consistency checking algorithm and the different model types as well as the integration of multiple learning algorithms. The framework is extensible with diverse types of models and learning algorithms in the Eclipse platform and publicly available.

In *ConsistAnts* tool, we provide two different types of probabilistic models: (1) probabilistic Rabin automata [125], (2) equivalent stochastic regular expressions [128]. The conversions between those models (Chapter 5), and the learning algorithms AAlergia [25], and pBLA (see Chapter 7.1.1) are also integrated to the tool.

The consistency checking algorithm works with a PRA and a trace dataset as an input. It reports our consistency result, which contains information on the degree of consistency and a list of counterexamples traces. *Trace dataset models* are lists of traces, each of which comprises of an ordered list of transition labels.

7.1.3.2 *The Consistency Checking Algorithm*

The consistency checking algorithm is based on the causal footprints as known from the process theory [3]. The original algorithm is used for the conformance checking of process models to validate the obtained models during process mining. Conceptually, we do not examine a complete path in an automaton or a complete trace but rather, only a pair of two transitions. The idea of causal footprints revolves around the *directly-follows-after* relationship between two transitions. Two characters a and b of an alphabet are said to directly follow another (b directly follows a) if:

- there is a state s such that there is an incoming transition with label a and an outgoing transition with label b, while both transitions have a non-zero transition probability for the **PRA** model.

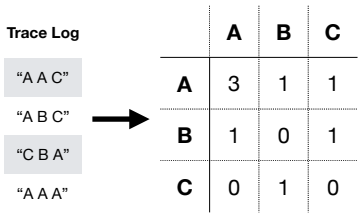


Figure 37: A trace dataset for the alphabet $\{A, B, C\}$ and its footprint matrix

- *b* follows directly after *a* anywhere in the trace for **trace datasets**.

Thus, a matrix is constructed based on this relationship. Each cell of this so-called conformance matrix indicates the directly-follows-after relationship between the two characters of the alphabet. Using the same example of the trace data set provided in Table 6, we construct its footprint matrix in Figure 37). Unlike the causal footprints idea in the process theory, where the structural conformance of an event log with the model is relevant, we adapt the idea of causal footprints to the probabilistic case. Instead of using special symbols in the cells of the footprint matrices, we choose to count the number of occurrences of a specific *directly-follows* relationship. With that approach, we aim to capture the statistical characteristics of the trace dataset and the PRA.

Below, we describe the three main-steps that our consistency checking algorithm performs. Finally, we also discuss how the resulting footprint matrices may be used to extract information on counterexamples in the trace dataset.

Computing Trace Log Footprint Matrices

For each trace of a given set of traces, we look at pairs of transitions by step-wise moving from the left to the right of a trace. For each pair of transitions, we then modify the matrix accordingly, as demonstrated in Figure 37.

After populating the footprint matrix with all available traces, we normalize the cell values by dividing them by the total number of processed traces. This process is required to compare them with the model footprints.

Computing Automata Footprint Matrices

All possible paths in the PRA model have to be explored to compute the footprint matrices for the model. However, we can limit this (possibly infinite) number of paths to explore by length due to the finite nature of the input trace dataset. Only paths up to the maximum trace length in the given trace dataset have to be explored; otherwise, there would be no traces to compare each other.

The computation of the footprint matrices can be recursively defined, as illustrated with the algorithm in Figure 38. While traversing the automata, the probability of the currently explored path is captured by the parameter `pathProbability`. For every explored pair of transitions, its respective probability value is added to the corresponding cell in the footprint matrix (note that for the sake of brevity, we omit the required special handling of transition pairs with the empty word as well as the recursion termination criteria for long paths).

```
Data: Probabilistic Automaton    Result: Footprint Matrix
Initialization: Visit(automaton.initialState, "",1.0)
def Visit(state, previousLabel, pathProbability) is
    for transition ∈ state.outgoing do
        currentPathProbability ← pathProbability * transition.probability;
        footprint[previousLabel][transition.label] += currentPathProbability;
        Visit(transition.target, transition.label, currentPathProbability)
    end
end
```

Figure 38: Computation of the automata footprint matrix

Comparing Footprint Matrices

The resulting footprint matrices for the traces and PRA, can now be compared by computing the difference. Our algorithm computes the footprints for subsets of all traces and compares them with corresponding automata footprints. These subsets T_n are created based on the length of the traces in the dataset and are specified as follows: $T_n := \{t \mid t \in traces \wedge t.length >= n\}$[3]

In accordance with this concept, we define $F_n(A)$ to represent the automaton footprint of paths up to a length of n and $F_n(T)$ the footprint gained from the trace subset T_n by only considering the first n transitions of each trace. The conformance matrix for a trace length t of an automaton model A with a trace dataset T is specified as: $C_n(A, T) := F_n(A) - F_n(T)$.

We form the average of all conformance matrix cells to reduce these conformance matrices to scalar values. This subdivision of conformance results on a per trace length basis aims to allow users to gain a more differentiated insight in the actual conformance of a model. At the conceptual level, it determines the conditional consistency assuming a minimum trace length.

7.1.3.3 *Counterexample Detection*

ConsistAnts suffices to compare footprints of the model and the trace dataset in order to detect counter examples. A cell with value "zero" in the PRA footprint indicates that the model does not allow that specific pair of transition to directly follow another. If, however, the trace dataset footprint yields a non-zero result for that cell, there must be a counterexample in the dataset. Thus, the algorithm searches the dataset for traces that contain exactly that pair of transitions to determine concrete counterexample traces. Probabilistic counter examples are, therefore, affect the degree of the consistency. The ConsistAnst framework provides the consistency ratio of the PRA model over the trace data.

7.1.4 *Using ConsistAnts for the Consistency Evaluation of pBLA*

Initially, we develop a generator that allows us to randomly generate automata models with corresponding trace datasets. Based on *Model Size* (number of states are in the size

[3] $traces$ and and $t.length$ represent the set of traces in a trace dataset and the number of transitions of a given trace t, respectively.

of the model), our generator outputs increasingly complex models and corresponding trace datasets (in terms of number of outgoing transitions). Figure 39 illustrates the execution time of the consistency checking on randomly generated inputs including PRA and traces. We can interpret the results as that "the checking time is not correlated with the model size" We explain these results by the working principle of the algorithm. For the computation of automata footprints, all paths in the model are traversed. Even though the *number of states* and the *number of transitions* are potential indicators for the model complexity, the number of paths is highly dependent on the structure of an automaton. Therefore, the actual structure may cause either a comparatively small or large number of exploratory paths.

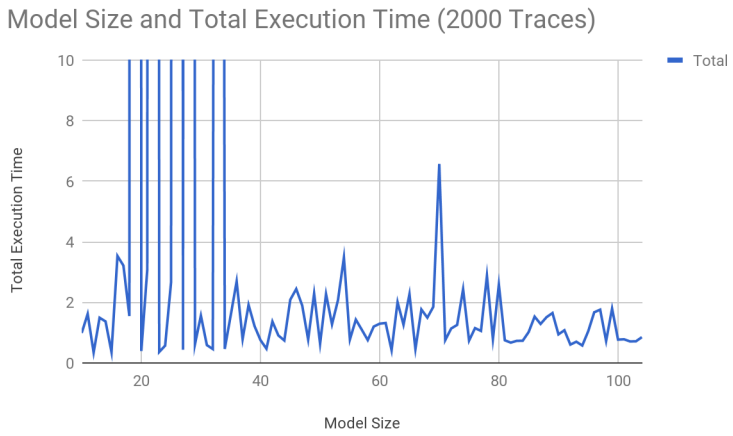
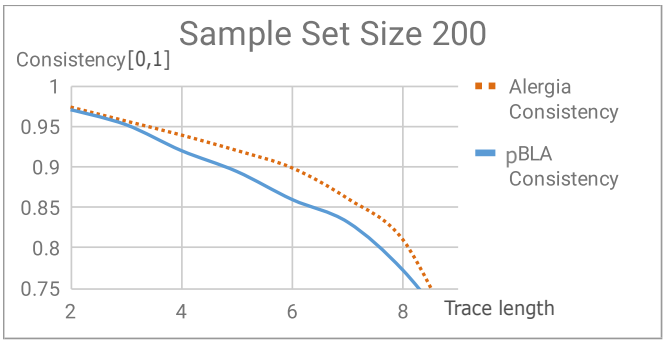


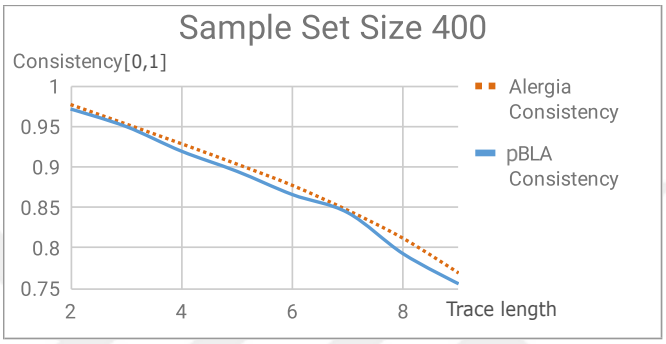
Figure 39: Execution time (seconds) of *ConsistAnts* for the random data

We run two different algorithms AAlergia and our probabilistic BLA on the same trace sets. The learned PRA through the AAlergia approach and the translated PRA from the learned SRE model through the pBLA models are provided as input models. Various sample sets are generated between 200-1980 lengths, and the consistency ratios of these input models over the sample trace sets are presented in Figure 40. While AAlergia produces higher consistency, pBLA has a close consistency in between 0.75-0.99. The average difference in the consistency rate between pBLA and AAlergia is presented in Figure 41. We consider average consistency for each trace length (from 2-9) and calculate the Pearson correlation [33] between AAlergia and pBLA. The analysis has resulted in 0.9994, which is a significant high correlation ($p < 0.01$).

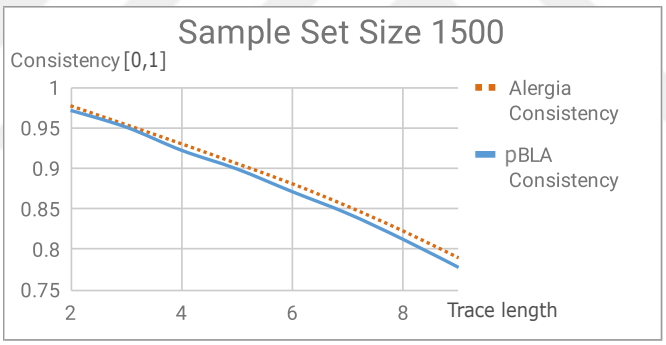
The framework also checks if a model has counterexamples against the trace data set. Our consistency checking algorithm described in the next section shows that our adapted pBLA produces correct results. In other words, it produces SRE models that do not generate counterexamples against the input sample set.



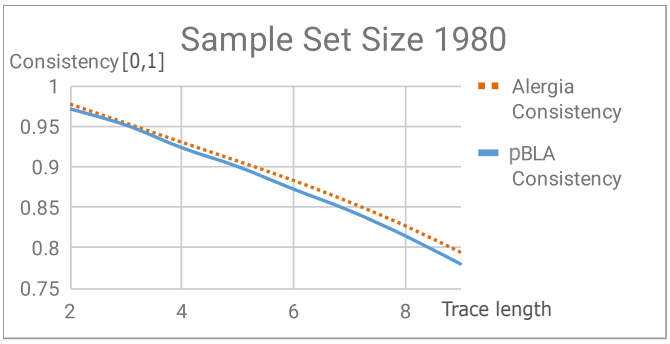
(a) Sample set: 200



(b) Sample set: 400



(c) Sample set: 1500



(d) Sample set: 1980

Figure 40: Consistency evaluation

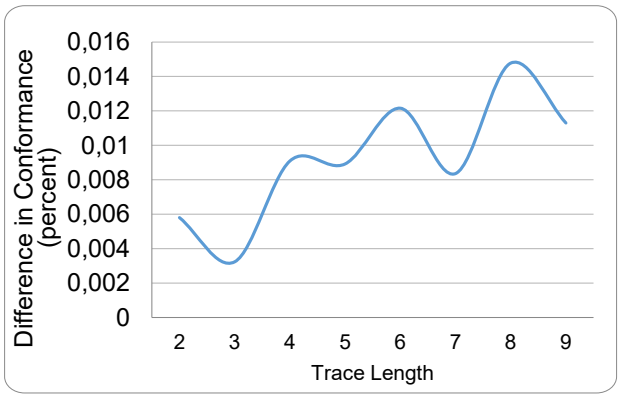
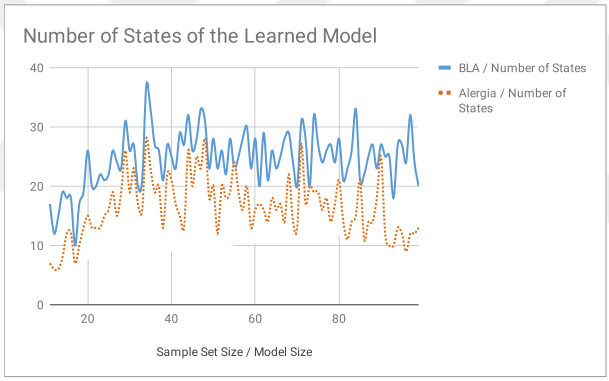


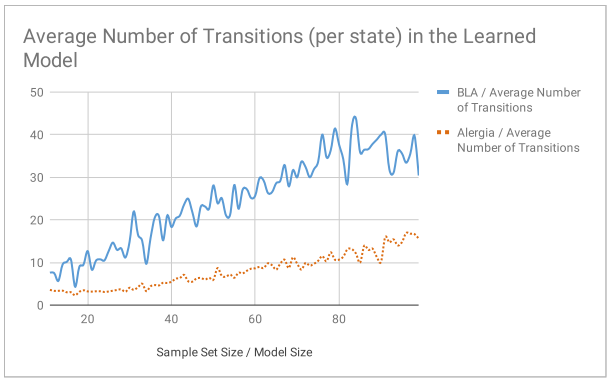
Figure 41: Average consistency difference between pBLA and AAlergia approaches in percentage

7.1.4.1 *Output Model Complexity*

Although the output models of both learning algorithms are different (PRA and SRE), it is possible to transform SREs to equivalent PRA and compare basic properties of the two automata.



(a) Model complexity in terms of number of states



(b) Model complexity in terms of number of transitions

Figure 42: Model complexity comparison between the learned PRA model and the translated PRA model from the learned SRE model

However, this evaluation is questionable, since model quality, which is not a numeric value to be measured, cannot only be assessed by the size and the complexity. Still, Figure 42 presents an interesting output: PRA models that are transformed from the

learned pBLA models tend to have more states than AAlergia models (Figure 42a), while the situation is opposite concerning the number of transition in the comparison (Figure 42b).

7.1.5 *Borg Case Study*

We have presented the consistency checking algorithm and applied it in the evaluation of the learning algorithm pBLA. In this section, the application and the evaluation of pBLA are extended with a case study based on the Google Borg Data Set [67].

Hence, we apply our consistency checking algorithm to the learned models of *AAlergia* [25] and pBLA (Section 7.1.1) respectively. We have measured the consistency of the resulting models with the original trace dataset. The assumption in this case is that the results of such learning algorithms are expected to yield high values in consistency.

We corroborate our validation with the data from the Google cluster management system called Borg [67]. The Google dataset is a collection of 29 days of task executions from a selected set of machines and is released for research purposes. Any allocated job on a worker machine includes multiple tasks that can execute multiple processes on the same machine. The life-cycle of a single job or task is provided in Figure 43 [141]. The dataset from Google is preprocessed, and concurrent tasks (over 3 jobs) are selected based on the time stamps in the log and finally converted to a trace dataset.

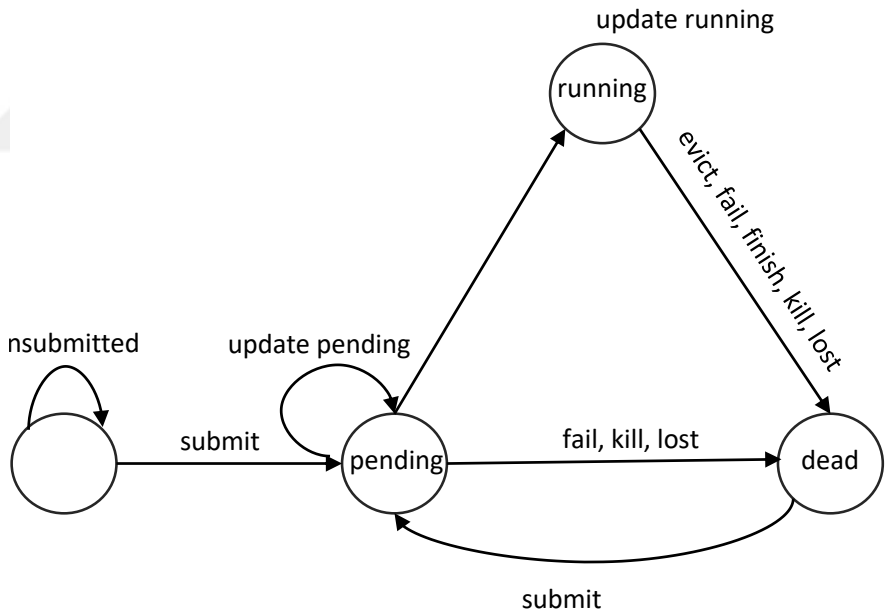


Figure 43: Lifecyle of the jobs and the tasks in Borg case study [141]

Figure 44 provides a time measurement over 100 traces compared to 1000 traces to get a glimpse of the *ConsistAnts'* execution time for a real application to check the consistency of PRA models learned by AAlergia algorithm. We can observe a tiny variation in the execution time for different sizes of the trace datasets.

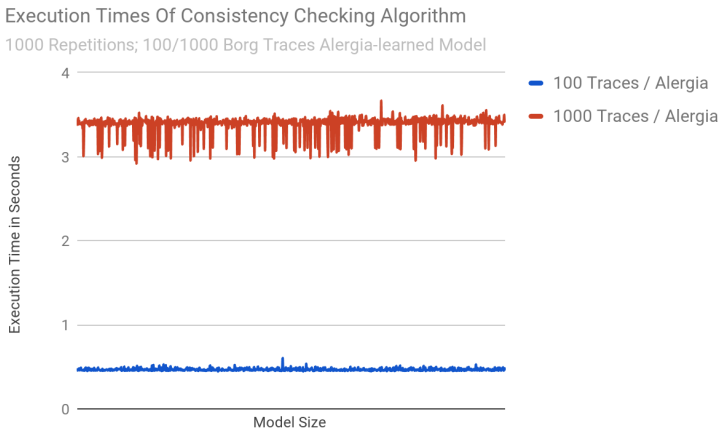


Figure 44: Execution time of *ConsistAnts* for the Borg case study

Another important observation is that the consistency of the models from AAlergia can get close to 100% in Table 10, if we have enough data in the trace log, i.e., with similar numbers of traces per trace length (AAlergia models yield slightly more consistent models compared to pBLA [④]). An inverse proportion between trace length and consistency becomes apparent. We find this result a quite logical consequence as the size of the sample taken from the trace dataset shrinks (subsets in Section 7.1.3.2).

Overall, both learned models from AAlergia and pBLA exhibit consistency values of over 98,5%.

	100 Traces		1000 Traces	
Trace Length	Alergia	BLA	Alergia	BLA
2	99.99	99.82	100	99.82
3	99.98	99.69	99.99	99.69
4	99.95	99.49	99.99	99.49
5	99.93	99.28	99.99	99.28
6	99.89	99.16	99.97	99.16
7	99.88	98.97	99.97	98.98
8	99.83	98.76	99.95	98.77
9	99.78	98.59	99.93	98.61
Avg.	99,90	99,22	99,97	99,22

Table 10: Consistency results (%) for the Google Cluster data set

Summary

The application of our consistency checking algorithm shows that our adapted learning method **pBLA** is correct in a way that it does not produce SRE models that lead to counterexamples with the input sample set, which means that the sample set is fully-contained in the learned language. The Performance Evaluation (see Figure 35)

④ Note that we applied minimization to the transformed probabilistic automata models to achieve a fair comparison (the probabilistic automata and stochastic regular expressions are known to be equivalent (See Chapter 5)).

displays that pBLA runs in linear time compared to AAlergia's quadratic time complexity. Due to the nature of the generalization technique used in pBLA learning, it is not parametrizable as AAlergia is. AAlergia provides the parameter that specifies a threshold for the computation of probabilistic similarity, but pBLA provides no such configurability, which can be seen as a weakness.

However, the pBLA learning algorithm might be evolved by adopting Fernau's branch handling [47] (See step 3 of the introduction of the original method in this chapter). Furthermore, it might be of interest to change the computation of the repetition probability in step 3 of the pBLA learning method.

On the other hand, we have developed a consistency checker between models and usage profiles or event logs by adding a stochastic aspect to the footprint algorithm to evaluate and compare the consistency of learning algorithms that is based on conformance checking. We aim to answer the following question by developing the generic framework *ConsistAnts*: Do quantitative models actually represent software behavior?

To answer this question, we have designed and implemented the footprint matrix algorithm by additionally considering the probabilities. Given the model of a software and traces of that software, it can determine relative degrees of consistency by trace length. We have achieved to produce meaningful consistency results in a reasonable execution time for most of the conventional model sizes.

Our evaluation process produces a set of indicators that can be used to distinguish models of higher and lower quality, meaning that this algorithm can be used to compare the degree of representation of the SRE model and software behavior. *Hence, we could indicate that our SRE modeling can be practically used for software models* (Model-Trace Consistency in Figure 34).

7.1.6 *Transformations from PRA to SRE*

Second way to obtain SRE models in practice is to apply transformations of PRA as described in Chapter 5. For this, we obtain SRE models by converting the existing probabilistic models publicly available in the PRISM probabilistic model benchmark [103]. The PRISM benchmark includes various case studies and probabilistic model types from DTMCs [9] to process algebras [30]. Even though there exist no PRA models explicitly, one can convert DTMCs into PRA by moving the state labels to transition labels [8]. One of the case study models describing a DTMC is leader election case study that we use for the transformation measurements. A short description of the Synchronous leader election case study is provided in the following paragraph.

Synchronous Leader Election Case Study

Leader election is designed to be a protocol for a synchronous ring of **N** processors such that they will be able to elect a leader by sending messages around the ring. The protocol proceeds in rounds, and the round is parametrised by a constant K. Each round begins with all processors choosing a random number (uniformly) from $1,...,K$ as an *id*. The processors then pass their ids around the ring. If there is a unique *id*, then the processor with the maximum unique id is elected the leader; otherwise, the processors begin a new round. We use the DTMC model from the PRISM benchmark that is modelled in the PRISM syntax based on the reactive modules [5], which are in the abstract level of modeling. One can construct and export explicit models (probabilistic

transition matrix) out of synchronized leader modules by the PRISM interface. Afterwards, we apply two methods to transform explicit models with various parameters (N,K) into SREs, whose theoretical foundations are provided in Chapter 5.

7.1.7 *Application of State Elimination Using Model Transformations*

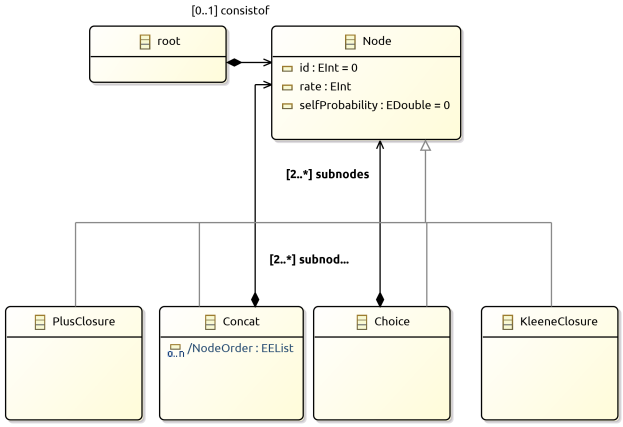
State elimination is a widely used technique using the translation between FSA and regex in practice. JFLAP [124] is an open source tool implemented in Java that encapsulates the most functionalities of the FSM, and regex translation, as well as their minimization. We have implemented a probabilistic version of JFLAP state elimination algorithm to get SREs. The performance evaluation of this translation is provided in Table 11.

n	k	states	transitions	JFLAP (sec)
3	2	26	33	0.13
4	2	61	76	0.31
3	3	69	95	0.68
5	2	141	172	6,86
3	4	147	210	8,56
3	5	273	397	97,16
4	3	274	354	100,47
6	2	335	398	227,47
3	6	459	674	852,99
4	4	812	1067	timeout
5	3	1050	1292	timeout

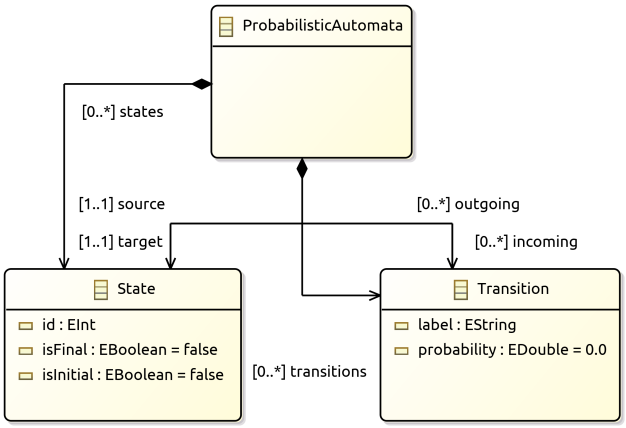
Table 11: Transformation measurements for JFLAP. Sizes of input models are indicated by the number of states and the transitions; execution times are reported in seconds.

We have presented PRA to SRE transformations as a model transformation problem in *Transformation Tool Contest* that is organized under International Conference on Model Transformation (ICMT) to increase the efficiency and to get larger models. We provide metamodels of PRA and SRE (demonstrated in Figure 45a and 45b, respectively) as well as the probabilistic state elimination algorithm (see the details in Chapter 5) in our case description [62]. In the scope of the tool contest, some solutions were proposed, such as the ones using Henshin [134], Epsilon [131], .NET ModelingFramework (NMF) [84], and SDMLib - Story Driven Modeling Library [144].

An example improvement is provided by using Henshin [135] transformations in the execution time performance [134], as demonstrated in table 12.



(a) SRE metamodel



(b) PRA metamodel

Figure 45: Metamodel of SRE languge and PRA using Eclipse Modelling Framework (EMF [42])

7.1.8 *Application of Brozowoski Method*

The other method is the probabilistic extension of Brozowski method whose proof is constructed in Chapter 5. The SRE tree structure is implemented with the same inheritance relationship as defined in the SRE metamodel (Figure 45a). An SRE node can be a type among of *Choice, Kleene, Plus closure, Concat*, and *Action*. Such a tree structure instead of string data enables to store larger SRE models and the memorization of probabilistic functions that are calculated on the tree nodes.

The measurements of the translation from the leader election DTMC explicit models (N and K are the parameters representing the number of processors and the round constant, respectively) into SRE models are presented in Table 13, which are more efficient to produce larger models.

				Transformation Time (sec)	
n	k	states	transitions	JFLAP	Henshin
3	2	26	33	0.09	0.21
4	2	61	76	0.14	0.25
3	3	69	95	0.49	0.29
5	2	141	172	3.46	0.78
3	4	147	210	4.37	0.77
3	5	273	397	58.60	2.93
4	3	274	354	57.78	2.32
6	2	335	398	143.12	3.45
3	6	459	674	461.64	10.09
4	4	812	1067	4786.58	48.15
5	3	1050	1292	timeout	timeout

Table 12: Transformation measurements for JFLAP and Henshin. Sizes of input models are indicated by the number of states, execution times are reported in seconds [134]

n	k	states	transitions	time	nodes	tokens
4	4	812	1066	3.7 ms	1786	7508
4	6	3962	5256	24.6 ms	9033	41984
4	8	12400	16494	95.7 ms	27741	63670
5	6	31382	39157	231 ms	60711	136970

Table 13: Transformation measurements for the leader election case study with Brozowoski algebraic method

7.1.9 *Comparing Verification Results with PRISM: Sanity Check*

We have discussed various approaches to obtain SRE models both from DTMC models and by learning to execute them as input models for our probabilistic model checking framework up to this section. Once we have SRE models of the leader election case study, we execute our reachability algorithm and the PRISM tool to check the properties $\mathcal{P}_{[1,1]}(\mathtt{True}\ \mathcal{U}\ (\mathcal{X}_{elected}\mathtt{True}))$ and $\mathcal{P}_{[0.8,1]}(\mathtt{True}\ \mathcal{U}^{\leqslant 10}\ (\mathcal{X}_{elected}\mathtt{True}))$ for the sanity check of the model checking results under the knowledge of the SRE-PRA equivalence. We have executed the PRISM object from a Java code which runs the hybrid engine as well as the explicit engine for the explicit models. The average results are demonstrated in Table 14 (the result occurs with probability 1.0; therefore, the outcome of the model checking is true) and Table 15 (the probability results are presented in the column "MC Result") are obtained by executing the PRISM explicit engine and our SRE tool for 20 times. The results of bounded probabilistic model checking for a DTMC and a SRE are almost same for leader election models since the SRE models are minimized in this example. However the results might differ for not minimized model of the equivalent SRE, whereas this case yield no difference for unbounded model checking.

PRISM constructs the model and transforms it into an MTBDD (Multi-terminal Binary Decision Diagrams) data structure [103]. Therefore, it might be unfair to compare total times as well as model checking times since PRISM engine already handles the

	Model Size			Execution time (ms)						
	SRE length	DTMC		PRISM			PRISM-Explicit			SRE Checker
Model (N,M)		Number of States	Number of Transitions	Const.+ Reach.	MC	**Total**	Const.+ Reach.	MC	**Total**	**Reach. +MC**
leader 3,2	207	26	33	16	0	16	14	0	14	1
leader 3,3	654	69	95	16	0	16	16	0	16	2
leader 3,4	1598	147	210	18	0	18	15	0	15	3
leader 3,6	5293	459	674	21	0	21	26	0	26	6
leader 3,8	13120	1059	1570	24	0	24	68	0	68	11
leader 4,2	457	61	76	16	0	16	15	0	15	1
leader 4,3	2495	274	354	22	0	22	20	0	20	5
leader 4,4	7508	812	1067	24	1	25	47	1	48	9
leader 4,5	19769	1933	2557	32	2	34	186	2	188	15
leader 4,6	41984	3962	5257	39	2	41	608	2	610	18
leader 5,2	1072	141	172	17	0	17	15	0	15	3
leader 5,3	8741	1050	1292	26	1	27	70	1	71	8
leader 5,4	39255	4244	5267	52	3	55	732	3	735	20
leader 5,5	126066	12709	15833	94	11	98	6645	4	6649	43
leader 6,2	2545	335	398	21	1	22	20	1	21	4
leader 6,6	2517774	234210	280865	2059	231	2290	out of time			1434

Table 14: Size and time measurements (ms) on the leader election case study models for the property $\mathcal{P}_{[1,1]}(\mathtt{True}\,\mathcal{U}(\mathcal{X}_{elected}\mathtt{True}))$ (Reach., Const. and MC stand for Reachability, Construction and Model Checking respectively).

reachability during the model construction, such that most of the pre-computation has already been performed before model checking phase while SREs do not require any reachability analysis. The measurements provide a sanity check, present the applicability of our approach, and relay the execution time for the existing models. We discuss the details and present the **incremental** Quantitative Verification results in the next sections.

7.2 INCREMENTAL VERSUS NON-INCREMENTAL APPROACH

This section evaluates the incremental quantitative verification framework through internal and external evaluations.

7.2.1 *Internal Evaluation*

We compare the incremental approach versus the non-incremental approach internally both on random data and synchronous leader election case study models. The *IQon* (Incremental Quantitative Verification) framework is developed on the top of a parser using SRE grammar attached with verification values. Once an initial model is parsed, the SRE tree is equipped with the verification results that store the maps of probabilistic values. The non-incremental approach constructs the tree from the scratch as well as calculates the attached probability functions, whereas the incremental approach calculates the propagated probability functions on the *affected* branch. From theoretical perspective, the complexity of the incremental approach will correspond to the traversal on the tree. Initially we ask the following research question:

Model (N,M)	Model Size			Execution time (ms)				**MC Result**	
	SRE length	DTMC		PRISM			SRE Checker	PRISM	SRE Checker
		Number of States	Number of Transitions	Const.+ Reach.	MC	**Total**	**Reach. +MC**		
leader 3,2	207	26	33	50	7	57	44	0.9375	0.9375
leader 3,3	654	69	95	32	2	34	15	0.98765	0.98765
leader 3,4	1598	147	210	33	2	35	20	0.99609	0.99609
leader 3,5	3042	273	397	31	3	34	19	0.9984	0.9984
leader 3,6	5293	459	674	36	2	38	36	0.99922	0.99922
leader 3,8	13120	1059	1570	33	3	36	63	0.99975	0.99975
leader 4,2	457	61	76	16	0	16	2	0.75	0.75
leader 4,3	2495	274	354	18	0	18	7	0.932784	0.932784
leader 4,4	7508	812	1067	26	3	29	31	0.975585	0.975585
leader 4,5	19769	1933	2557	30	5	35	67	0.989183	0.989184
leader 4,6	41984	3962	5257	69	8	77	136	0.994513	0.994513
leader 5,2	1072	141	172	15	1	16	3	0.3125	0.3125
leader 5,3	8741	1050	1292	23	3	26	20	0.740740	0.740740
leader 5,4	39255	4244	5267	56	12	68	182	0.878906	0.878906
leader 6,2	2545	335	398	16	1	17	7	0.1875	0.1875

Table 15: Bounded reachability analysis results on leader election case study models for the property $\mathcal{P}_{[0.8,1]}(\mathtt{True}\ \mathcal{U}^{\leqslant 10}\ (\mathcal{X}_{elected}\mathtt{True}))$

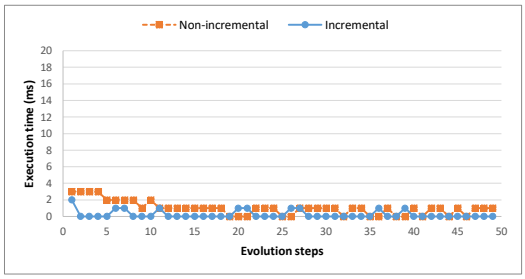
RQ1

Does the incremental verification (IQon) perform better than the non-incremental verification?

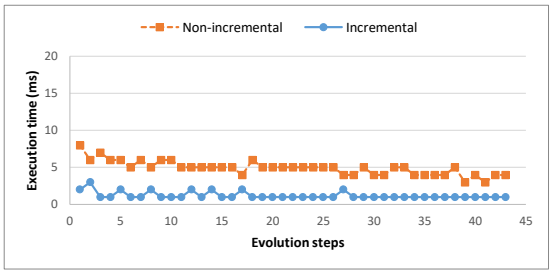
To answer this question, we generate different SRE models in different node sizes from 100 up to 1 million and apply applicable changes up to 5 edit operations for *one evolution step*. More specifically, change operations described in Chapter 6 are applied randomly on the SRE nodes based on the node type up to 50 steps (e.g., adding a sub-node to a *choice*, updating the probability in a *Kleene* node). The results are given for different data set as demonstrated in Figure 46. The SRE node size and the length represent the number of nodes in the constructed SRE tree, and the string length of the SRE, respectively. The comparison shows that the incremental verification rather outperforms when the model size increases.

Evolving Synchronous Leader Election Case Study

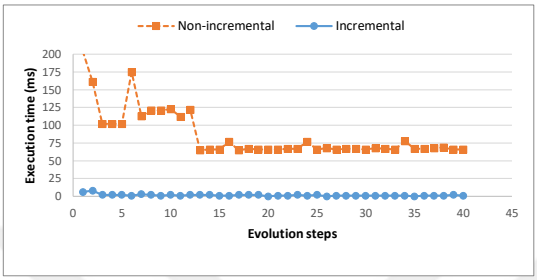
We show the incremental verification results by applying some evolution scenarios that are described as *edit operations* in the previous section 6. Initial models are the transformed models from leader election case study for different sizes produced via parameters (P,K). It is denoted as *leader P,K- node size* (e.g., leader 4,4-2135) in the figures showing the execution time measurements for 40 evolution steps (Figure 47a and Figure 47b. The execution time is much slower for the initial model since it is parsed and constructed. Then, we apply random changes on the constructed tree within the change operation up to 5 for *one evolution step*.



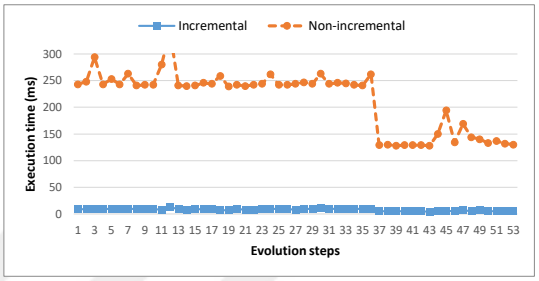
(a) SRE node size 100, length 803



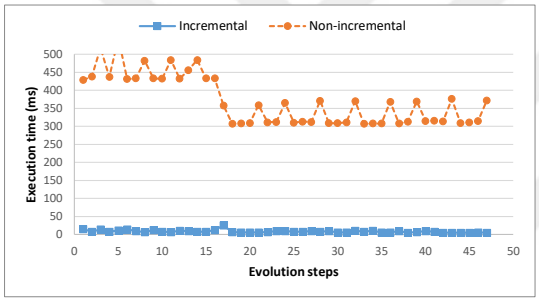
(b) SRE node size 1000, length 6395



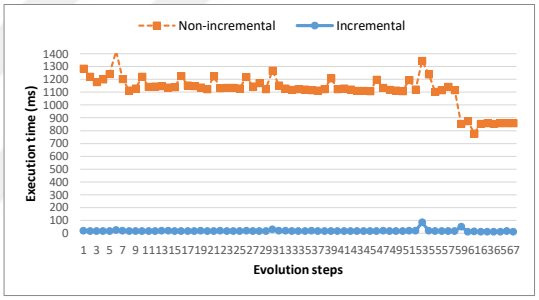
(c) SRE node size 10000, length 59872



(d) SRE node size 100000, length 609821

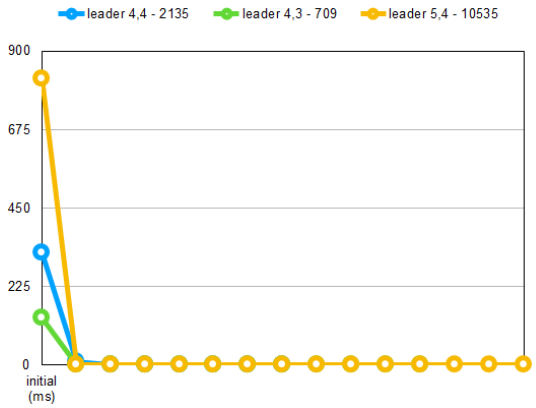


(e) SRE node size 500000, length 1228883

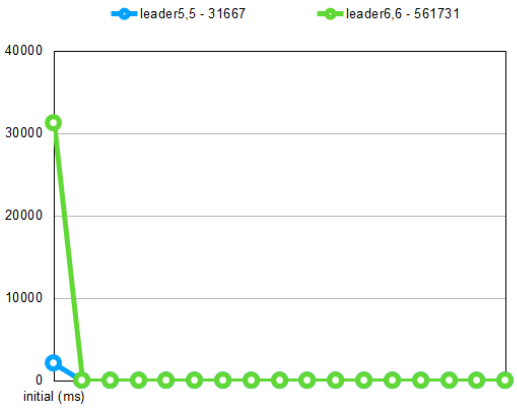


(f) SRE node size 1000000, length 3050138

Figure 46: RA$_1$.a) Random experiments on various data sets



(a) Leader election for the parameters (4,4) - (4,3) - (5,4).



(b) Leader election for the parameters (5,5) - (6,6)

Figure 47: RA$_1$.b) Leader election case study demonstrating execution time in milliseconds for 40 evolution steps and application of 5 edit operations for each evolution step.

RA$_1$

The incremental approach outperforms the non-incremental version. Furthermore, the benefit of the incremental approach increases as the input model size grows both in random models (Figure 46) and leader election models (Figure 47).

The investigation of the change size is an important parameter to reveal the efficiency of the incremental approach. Therefore, the second critical point is addressed with the following question:

RQ$_2$

How *change portion* and *change complexity* affect the incremental quantitative verification performance?

In this experiment, we narrow the node size in the range of 100,000-1 million and cluster the change rates by applying multiple edit operations (up to 100) (multiple *change size*) in one step. It affects the bigger parts of the models in different change rates. Thus, the affected parts (revisited nodes) determines the *change complexity*.

RA$_2$

The incremental/non-incremental ratio results show that the ratio for execution time does not increase rapidly even by the growth change (see Figure 48).

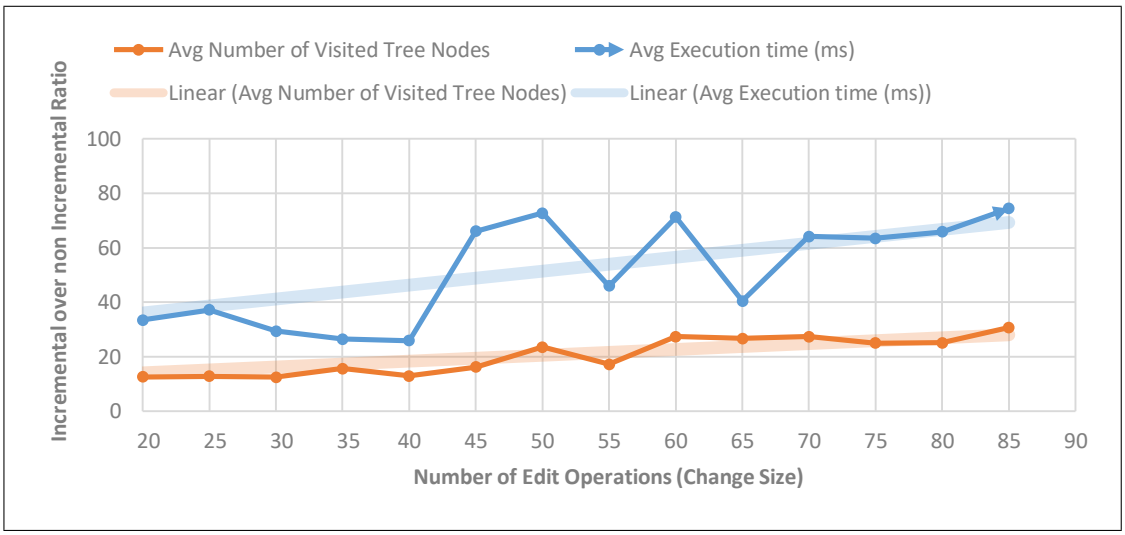


Figure 48: RA$_2$) Comparison of the incremental and non-incremental quantitative verification based on the change size

7.2.2 *External Evaluation*

This section evaluates the incremental quantitative verification (*IQon*) by comparing the-state-of the art probabilistic verification tools to discuss the generalizability of the proposed approach. Therefore, the following research question is deemed valuable.

RQ3

How efficient is *IQon* when compared to the-state-of the art tools for changing models?

In the previous subsections, we have mentioned PRISM, which is an established and well-known tool, and discussed the experiments we have collected. However, for the time comparison we have used the tool **Storm**[38], which is recently released and has been implemented on the top of the PRISM model checking. There are two reasons to choose the tool:

1. It is a recent and performance efficient tool.
2. It enables to use the explicit model as an input model for the model checking without describing a high level language, such as *Reactive modules*. Hence, we could apply the changes directly on the explicit model (probability transition matrix).

We have studied the change scenarios on a well-know case study called Tele Assistance System (TAS), established in the self adaptive software engineering research [1]. A brief description of the system is provided below.

Case Study in Self-Adaptive Systems: TAS

The tangible application Tele Assistance System (TAS) offers a smart health support to patients using home devices. TAS offers a composite service that is a combination of the following services [23]:

- Alarm Service, which provides the operation sendAlarm,
- Medical Analysis Service, which provides the operation analyzeData ,
- Drug Service, which provides the operations changeDoses and changeDrug.

A detailed description of the system can be found at [1]. A DTMC model of TAS is depicted in Figure 49. Since the original model is parametric (the transition probabilities are the symbols) [23], the model in 49 is adapted by assigning arbitrary probabilities to the transitions and some *changes* are applied (dotted shapes in green).

For realistic and large models, we set up models one up to nine TAS systems by modeling them as interleaving components using the PRISM syntax and the tool. We export the explicit models out of the interleaved models and implement a labeling mapping using the PRISM code to trace the state and labels together with the module they belong to (Figure 50). As a consequence, a change in a module is able to be automatically detected and propagated to the explicit model. For instance, an update in state s_0 in Module 1 (M_1) will lead to another update the change in state 0 and 1 (states labels are $M_1S_0M_2S_0$ and $M_1S_0M_2S_n$). Finally, the state labelled automata is renamed as transition labeled as shown in Figure 50 as an input for the transformation of it into an SRE.

In the sequel, we apply transformations described in Chapter 5 on the explicit models to obtain *equivalent* SREs.

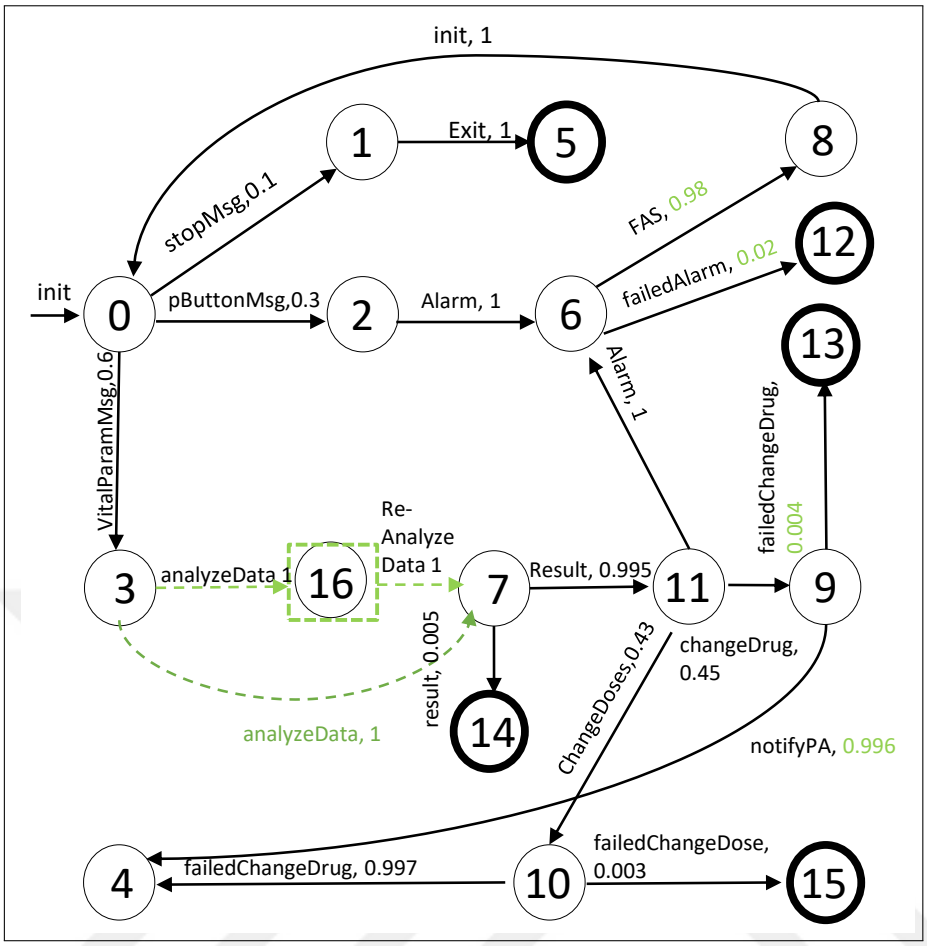


Figure 49: *Changing* DTMC model of TAS

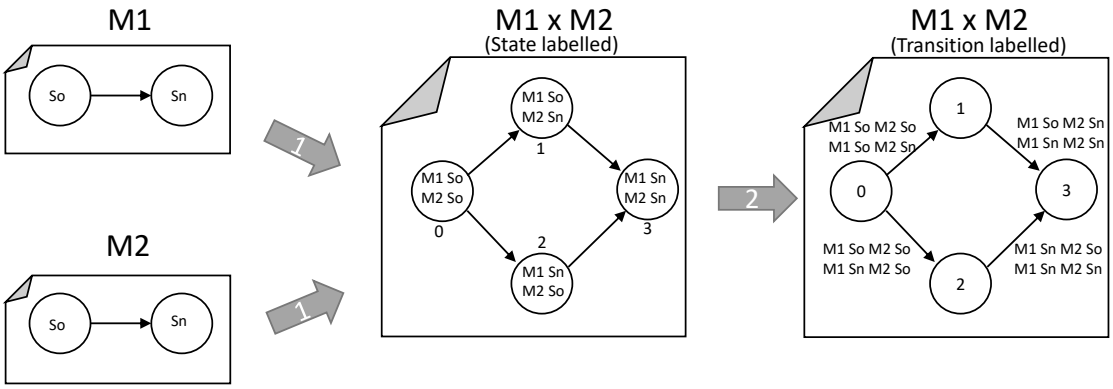


Figure 50: 1. Labeling of automata for module-state relationship. 2. Conversion of DTMC explicit model (state labeled) to a PRA (transition labeled)

We trace the relationship between probabilistic automata and SREs by using **Edge ⇔ SRE node** mapping whose incremental transformations are provided in Chapter 6.4. Consequently, we could apply the *equivalent change* in the corresponding SRE for each step, as demonstrated in Figure 51. The measurements are captured each time when the change is applied on the models.

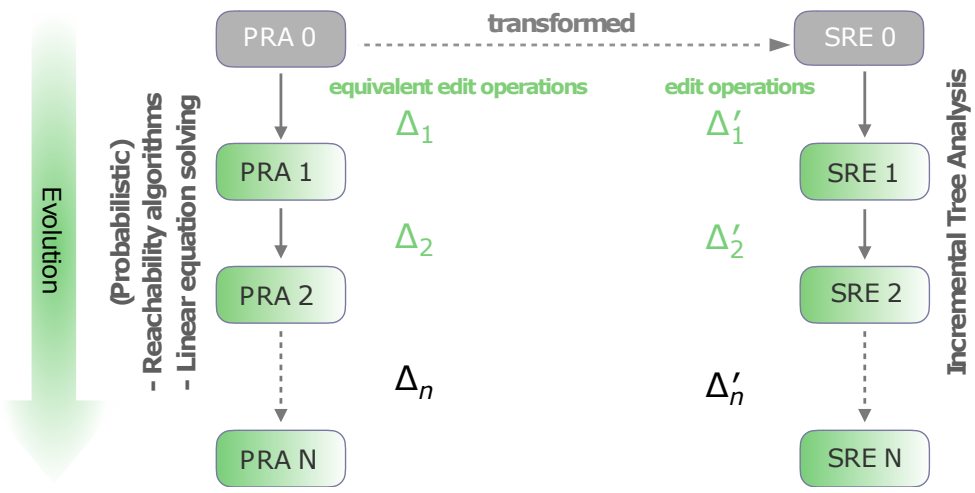


Figure 51: The experimental setup for the comparison of evolving models and model checking algorithms

Possible Scenarios

On the guidance of generic scenarios for service-based systems discussed in [145], we specify possible scenarios for TAS and apply some change patterns introduced in Chapter 6.

Scenario 1: In the failing of some services, e.g., alarm service, the system behaves with different probabilistic distributions since a service might be added or removed. Changing probabilistic distribution is a common evolution scenario for probabilistic systems. Especially in the scope of parametric model checking, the probabilities are assigned to variables and processed in the analysis as long as they are extracted from the run-time data. Our framework can be also adapted to the parametric model checking approach. Nevertheless, in this step of the evaluation, we apply the *change* as an *update* operation of the distribution for the evolution steps.

RA3

Initial comparison based on the average execution times indicates that our IQon framework accelerates as the model size gets larger (see Table 16). Although the Storm is highly efficient in general, it constructs the model and runs the analysis from scratch even for small changes.

Change size might alter the outperformance trends; however, our implementation has been improved with a hashmap that keeps all the intermediate SRE tree nodes and the calculations. Such improvement speeds up the execution time with the hashmap hit up to %60.

Scenario 2: We use *adding a task* pattern in the initial scenario. Assume that in the process of data, "re-analysis" of the data is required to be double-checked for some critical cases. Such an action exists between state 3 and 7. We insert another action and a new state (16) as shown in Figure 49. Note that the aim of this pattern is to add one step task rather than a loop.

Execution Time (ms)	IQon		Storm	
	Initial	**Evolution**	Initial	**Evolution**
TeleAsistant (1)	263	0	0	0
TeleAsistant (2)	278	0	0	1
TeleAsistant (5)	375	**2**	3	3
TeleAsistant (8)	1415	***12***	101	88
TeleAsistant (9)	1589	***29***	440	443

Table 16: RA_3: External evaluation for changing models

Applying this change in one module (one TAS component) might affect the whole explicit model since they are interleaving. We could identify the states to update using the labelling approach 50.

A change in a state will affect the model in the size of $m^r + n^r$, where m is the number outgoing, n is the incoming transitions from state s, and r is the number interleaving components (modules). Adding a new state between state 3 and 7 affects two rows of transition matrix in the explicit model whereas adding a state between 7 and 11 affects 2^9 rows in the explicit model. Although this number seems to be big, it is still a small portion of the whole model in the size of 40000 rows. The execution time for such an update remains minor (1-10 milliseconds) for diverse sizes of models in *IQon*. Such a change is not applicable for Storm input models.

7.3 CONCLUSION

We have implemented an incremental quantitative verification framework for stochastic regular expressions (SRE) formally founded in Chapter 4. We have offered two techniques to obtain SRE models: (1) probabilistic BLA (pBLA) - learning SREs from the trace data set (2) applying transformations from PRA models. These techniques are means to prepare the data set to evaluate our SRE checker and the incremental quantitative verification framework (*IQon*).

In the phase of obtaining models, we test the results with respect to the validity of the introduced SRE learning technique by using a consistency checker called *ConsistAnt* based on a footprint matrix algorithm in the process theory. Thus, we could also show the practical use of SRE modeling representing the software behaviour.

Moreover, the SREs translated from the PRA models are processed into the SRE checker, and the findings (both time measurements and model checking results) are compared with a widely used probabilistic model checker PRISM as a sanity check. Hence, we are able to provide the consistency of the model checking results with PRISM. In this stage of the comparison, we list some difficulties and threats to validity items as follows:

1. The SRE checker and the conventional probabilistic model checkers PRISM or Storm encapsulate different techniques with possible centric algorithms, which brings the difficulty to compare the measures. In other words, while SREs are analyzed based on string searches, the traditional probabilistic model checking applies a model construction together with reachability, bisimulation reductions, and linear equation solving.

2. The size of the models might be irrelevant depending on their design [69].

3. There exist no familiar approach applying model checking on SREs. Furthermore, SREs are jet another modeling language comparing to Markov chains and automata even though they are known to be equivalent.

We apply both internal and external evaluations for the *IQon* framework (Chapter 6) implemented on the SRE checker (Chapter 4). The internal evaluation presents the comparison of incremental vs non-incremental approach of the SRE checker itself applying edit operations described in Chapter 6. The results report that the incremental approach (IQon) outperforms by far the non-incremental one. In the worst case, the applied change might traverse the whole tree.

We use a recent and efficient tool *Storm* for the external evaluation owing to its feature to use an explicit models as an input. This feature enables us to apply changes directly on the explicit models and gather the measurements. During the comparison with *IQon*, our mapping approach guides for a fair comparison to apply the corresponding changes for both model type SRE and PRA as input models for *IQon* and *Storm*, respectively. The comparison is applied on a well-known case study model for evolving and adaptive systems called TAS. The process reveals that *IQon* outperforms Storm as the model size increases for multiple edit operations. Hence, the fundamental motivation of this comparison is successfully and ultimately evaluated for *IQon* while *today's software is continuously updated with tiny changes for large models*.

The limitation of *IQon* still subsists on designing and getting SRE models in practice. Also, the identification and optimization of the changes is still open to discussion for the probabilistic models during the incremental verification.

Part V

CONCLUSION

8 CONCLUSION AND FUTURE WORK

The need for *incremental verification* has been recently studied as a far-reaching problem of the changing software [64]. We have captured the problem of efficient quantitative verification in the scope of evolving systems.

This thesis presents a novel approach for the incremental verification of probabilistic systems underlying a mathematical framework. One of the main contributions of this thesis is a formalism for model checking probabilistic properties with SRE trees. Hence, we use the benefits of the modularity of such trees for the incremental verification. First of all, we provide the equivalence between stochastic regular expressions and probabilistic Rabin automata. A similar language to SREs and their equivalence to probabilistic Rabin automata is provided in [19]. The formalism we have proposed is similar Kleene-like languages and automata trees. The idea of model checking with words lies on the LTL logic [18]. However, we seek to check if disjoint sets of words are satisfying the formula with a probabilistic branching logic.

We describe a translation of the PACTL word formulas into SREs and formalize the semantics as a satisfaction relation between the words of the SREs and PACTL formulas. The essential idea is lying on this semantics, where we find the meaning of $w \models E_p$ such that w is a word of the SRE and E_p is a `propertySRE`, which is a PACTL formula in the SRE form. When constructing the SRE node during parsing, we calculate the word sets and the fixpoints of these sets to check if they satisfy the E_p. We propagate the probabilities of satisfying words with hash functions to the parent node. The E_p is recursively checked on a parse tree, and the algorithm terminates when the tree is traversed bottom up.

Another contribution of this thesis is incremental quantitative verification *IQon* which is established on an SRE formalism for evolving probabilistic models. The idea of the IQon framework on the top of the SRE formalism is founded on the calculation of all information locally on every SRE term and generating solutions regarding the model checking property. We present a complete edit operations over the SRE syntax to apply change scenarios. Moreover, we identify some domain-specific change patterns to increase the applicability for various evolution scenarios in the long run of the software.

We present an evaluation framework of our approach that guides how to obtain SRE models for the analysis and shows how efficient our formalism (internal and external) is during the evolution. Finally, the validation is provided how consistent the SRE models are with the system behavior are. The evolution scenarios can be investigated comprehensively in the future work for real applications in the sense of type and identification of changes. Hence, the change patterns can be optimized and applied to observe the evolution steps.

We believe that our proposed framework can be extended to a concurrent representation of stochastic regular expressions. As future work, SRE trees can be extended with concurrency for probabilistic verification [54, 113].

Different application domains, such as query language generations, analysis of neural networks, string manipulations with probabilistic reasoning and model counting, are also other research directions.

BIBLIOGRAPHY

[1] https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/tas/. Online; last access December 2020.

[2] Horizon 2020. *Horizon 2018-2020 Work programme*. https://ec.europa.eu/research/participants/data/ref/h2020/wp/2018-2020/main/h2020-wp1820-fet_en.pdf. Online; last access December 2020. 2018-2020.

[3] Wil MP Van der Aalst. "Process Mining: Discovery, Conformance and Enhancement of Business Processes." In: *Springer-Verlag* 8 (2011), p. 18.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006. ISBN: 0321486811.

[5] Rajeev Alur, Thomas A. Henzinger. "Reactive Modules." In: *Form. Methods Syst. Des.* 15.1 (July 1999), pp. 7–48. ISSN: 0925-9856. DOI: 10.1023/A:1008739929481.

[6] Dana Angluin. *Identifying languages from stochastic examples*. Tech. rep. New Haven, CT: Yale University Dept. of Computer Science, 1988.

[7] Marco Autili, Lars Grunske, Markus Lumpe, Patrizio Pelliccione, Antony Tang. "Aligning Qualitative, Real-Time, and Probabilistic Property Specification Patterns Using a Structured English Grammar." In: *IEEE Transactions on Software Engineering* 41 (2015). DOI: 10.1109/TSE.2015.2398877.

[8] Christel Baier, Lucia Cloth, Boudewijn R. Haverkort, Matthias Kuntz, Markus Siegle. "Model Checking Markov Chains with Actions and State Labels." In: *IEEE Trans. Software Eng.* 33.4 (2007), pp. 209–224. DOI: 10.1109/TSE.2007.36.

[9] Christel Baier, Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. ISBN: 026202649X, 9780262026499.

[10] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, Marta Simeoni. "Model-Based Performance Prediction in Software Development: A Survey." In: *IEEE Trans. Software Eng.* 30.5 (2004), pp. 295–310.

[11] Steffen Becker, Lars Grunske, Raffaela Mirandola, Sven Overhage. "Performance Prediction of Component-Based Systems- A Survey from an Engineering Perspective." In: *Architecting Systems with Trustworthy Components*. Vol. 3938. Lecture Notes in Computer Science(LNCS). Springer, 2006, pp. 169–192.

[12] Steffen Becker, Heiko Koziolek, Ralf Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22.

[13] Simona Bernardi, José Merseguer, Dorina C. Petriu. "Adding Dependability Analysis Capabilities to the MARTE Profile." In: *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS*. Ed. by Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, Markus Völter. Vol. 5301. Lecture Notes in Computer Science. Springer, 2008, pp. 736–750. ISBN: 978-3-540-87874-2.

[14] Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, Antinisca Di Marco, Antonino Sabetta. "Towards a Model-Driven Infrastructure for Runtime Monitoring." In: *Software Engineering for Resilient Systems - Third International Workshop, SERENE 2011*. Ed. by Elena Troubitsyna. Vol. 6968. Lecture Notes in Computer Science. Springer, 2011, pp. 130–144.

[15] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, Dino Mandrioli. "Incremental Syntactic-Semantic Reliability Analysis of Evolving Structured Workflows." In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*. 2014, pp. 41–55. DOI: 10.1007/978-3-662-45234-9_4.

[16] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, Dino Mandrioli. "Syntactic-semantic incrementality for agile verification." In: *Sci. Comput. Program.* 97 (2015), pp. 47–54. DOI: 10.1016/j.scico.2013.11.026.

[17] Henrik C. Bohnenkamp, Peter van der Stok, Holger Hermanns, Frits W. Vaandrager. "Cost-optimization of the IPv4 zeroconf protocol." In: *2003 International conference on dependable systems and networks*. IEEE Computer Society Press, 2003, pp. 531–540. ISBN: 0-7695-1952-0.

[18] Bernard Boigelot, Axel Legay, Pierre Wolper. "Omega-Regular Model Checking." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Kurt Jensen, Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 561–575.

[19] Benedikt Bollig, Paul Gastin, Benjamin Monmege, Marc Zeitoun. "A Probabilistic Kleene Theorem." In: *Automated Technology for Verification and Analysis*. Ed. by Supratik Chakraborty, Madhavan Mukund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 400–415. ISBN: 978-3-642-33386-6.

[20] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, Mary Shaw. "Engineering Self-Adaptive Systems through Feedback Loops." In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee. Springer Berlin Heidelberg, 2009, pp. 48–70. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_3.

[21] Janusz A. Brzozowski. "Derivatives of Regular Expressions." In: *J. ACM* 11.4 (Oct. 1964), pp. 481–494. ISSN: 0004-5411. DOI: 10.1145/321239.321249.

[22] Doina Bucur. *Automata-based model checking for LTL*. http://doina.net/AR15/AR15-L45.pdf. Online; last access December 2020. 2015.

[23] Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, Giordano Tamburrelli. "Dynamic QoS Management and Optimization in Service-Based Systems." In: *IEEE Trans. Software Eng* 37.3 (2011), pp. 387–409.

[24] Radu Calinescu, Colin Paterson, Kenneth Johnson. "Efficient Parametric Model Checking Using Domain Knowledge." In: vol. 47. 6. 2021, pp. 1114–1133. DOI: 10.1109/TSE.2019.2912958.

[25] Rafael C Carrasco, José Oncina. "Learning stochastic regular grammars by means of a state merging method." In: *International Colloquium on Grammatical Inference*. Springer. 1994, pp. 139–152.

[26] Jean-Marc Champarnaud, Georges Hansel. "AUTOMATE, a computing package for automata and finite semigroups." In: *Journal of Symbolic Computation* 12.2 (1991), pp. 197–220.

[27] Betty H. C. Cheng et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap." In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. Lecture Notes in Computer Science. Springer, 2009, pp. 1–26. ISBN: 978-3-642-02160-2.

[28] Andrea Ciancone, Antonio Filieri, Mauro Luigi Drago, Raffaela Mirandola, Vincenzo Grassi. "KlaperSuite: An Integrated Model-Driven Environment for Reliability and Performance Analysis of Component-Based Systems." In: *Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011*. Ed. by Judith Bishop, Antonio Vallecillo. Vol. 6705. Lecture Notes in Computer Science. Springer, 2011, pp. 99–114.

[29] Frank Ciesinski, Christel Baier, Marcus Größer, Joachim Klein. "Reduction Techniques for Model Checking Markov Decision Processes." In: *Fifth Int. Conference on the Quantitative Evaluaiton of Systems (QEST 2008)*. 2008, pp. 45–54. DOI: 10.1109/QEST.2008.45.

[30] Allan Clark, Stephen Gilmore, Jane Hillston, Mirco Tribastone. "Stochastic Process Algebras." In: *Formal Methods for Performance Evaluation: 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures*. Ed. by Marco Bernardo, Jane Hillston. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 132–179.

[31] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu andHelmut Veith. "Counterexample-Guided Abstraction Refinement." In: *Computer Aided Verification*. 2000, pp. 154–169.

[32] Edmund M. Clarke, Orna Grumberg, Doron Peleg. *Model Checking*. MIT press, 1999. ISBN: 9780262032704.

[33] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 1988. ISBN: 9780805802832.

[34] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Christof Löding, Denis Lugiez, Sophie Tison, Mark Tommasi. *Tree Automata Techniques and Applications*. Available on: http://www.grappa.univ-lille3.fr/tata. Online; last access December 2020.

[35] Russ Cox. *Regular Expression Matching Can Be Simple And Fast*. https://swtch.com/~rsc/regexp/regexp1.html. Online; last access December 2020.

[36] Conrado Daws. "Symbolic and Parametric Model Checking of Discrete-time Markov Chains." In: *Proceedings of the First Int. Conference on Theoretical Aspects of Computing*. ICTAC'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 280–294. ISBN: 3-540-25304-1, 978-3-540-25304-4. DOI: 10.1007/978-3-540-31862-0_21.

[37] Rocco De Nicola. "Action and State-based Logics for Process Algebras." In: *CONCUR '91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26-29, 1991, Proceedings*. 1991, pp. 20–22. DOI: 10.1007/3-540-54430-5_77.

[38] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, Matthias Volk. "A Storm is Coming: A Modern Probabilistic Model Checker." In: *Computer Aided Verification*. Ed. by Rupak Majumdar, Viktor Kunčak. Springer International Publishing, 2017, pp. 592–600.

[39] Antinisca DiMarco, Claudio Pompilio, Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, Antonino Sabetta. "Yet another meta-model to specify non-functional properties." In: *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications, QASBA 2011*. ACM International Conference Proceeding Series. ACM, 2011, pp. 9–16. ISBN: 978-1-4503-0826-7.

[40] Ding-Shu Du, Ker-I Ko. *Problem Solving in Automata, Languages, and Complexity*. John Wiley and Sons, 2001.

[41] Devdatt P. Dubhashi, Alessandro Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009. DOI: 10.1017/CBO9780511581274.

[42] *EMF*. https://www.eclipse.org/modeling/emf/. Online; last access December 2020.

[43] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, Giordano Tamburrelli. "Model Evolution by Run-Time Parameter Adaptation." In: *International Conference on Software Engineering, ICSE 2009*. Ed. by Stephen Fickas, Joanne Atlee, Paola Inverardi. Vol. 4839. IEEE, 2009, pp. 111–121. ISBN: 978-1-4244-3452-7/09/.

[44] Kousha Etessami, Alistair Stewart, Mihalis Yannakakis. "Stochastic Context-Free Grammars, Regular Languages, and Newton's Method." In: *Automata, Languages, and Programming*. Ed. by Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, David Peleg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 199–211.

[45] William Feller. *An introduction to probability theory and its applications*. John Wiley and Sons, 2008.

[46] Lu Feng, Marta Kwiatkowska, David Parker. "Automated Learning of Probabilistic Assumptions for Compositional Reasoning." In: *Fundamental Approaches to Software Engineering*. Ed. by Dimitra Giannakopoulou, Fernando Orejas. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 2–17. ISBN: 978-3-642-19811-3.

[47] Henning Fernau. "Algorithms for learning regular expressions." In: *International Conference on Algorithmic Learning Theory*. Springer. 2005, pp. 297–311.

[48] Antonio Filieri, Carlo Ghezzi, Giordano Tamburrelli. "Run-time efficient probabilistic model checking." In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011, pp. 341–350. DOI: 10.1145/1985793.1985840.

[49] Antonio Filieri, Lars Grunske, Alberto Leva. "Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models." In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 2015, pp. 200–211. DOI: 10.1109/ICSE.2015.41.

[50] Vojtech Forejt, Petr Jancar, Stefan Kiefer, James Worrell. "Language equivalence of probabilistic pushdown automata." In: *Inf. Comput.* 237 (2014), pp. 1–11. DOI: 10.1016/j.ic.2014.04.003.

[51] Vojtech Forejt, Marta Z. Kwiatkowska, David Parker, Hongyang Qu, Mateusz Ujma. "Incremental Runtime Verification of Probabilistic Systems." In: *RV*. 2012, pp. 314–319.

[52] Vojtěch Forejt, Marta Kwiatkowska, David Parker, Hongyang Qu, Mateusz Ujma. "Incremental Runtime Verification of Probabilistic Systems." In: *Runtime Verification*. Ed. by Shaz Qadeer, Serdar Tasiran. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 314–319.

[53] Paul Gainer, Ernst Moritz Hahn, editor="Annabelle McIver Schewe Sven", Andras Horvath. "Incremental Verification of Parametric and Reconfigurable Markov Chains." In: *Quantitative Evaluation of Systems*. Springer International Publishing, 2018, pp. 140–156. ISBN: 978-3-319-99154-2.

[54] Vijay K. Garg, Ratnesh Kumar, Steven I. Marcus. "A probabilistic language formalism for stochastic discrete-event systems." In: *IEEE Transactions on Automatic Control* 44.2 (1999), pp. 280–293. ISSN: 0018-9286. DOI: 10.1109/9.746254.

[55] Simos Gerasimou. "Runtime quantitative verification of self-adaptive systems." PhD thesis. University of York, UK, 2016.

[56] Simos Gerasimou, Radu Calinescu, Giordano Tamburrelli. "Synthesis of probabilistic models for quality-of-service software engineering." In: *Autom. Softw. Eng.* 25.4 (2018), pp. 785–831. DOI: 10.1007/s10515-018-0235-8.

[57] Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, Matthias Tichy. "Supporting semi-automatic co-evolution of architecture and fault tree models." In: *Mendeley Data* v1 (2018). DOI: http://dx.doi.org/10.17632/6kkh54xbpj.1.

[58] Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, Matthias Tichy. "Supporting semi-automatic co-evolution of architecture and fault tree models." In: *Journal of Systems and Software* 142 (2018), pp. 115–135. DOI: 10.1016/j.jss.2018.04.001.

[59] Sinem Getir, André Van Hoorn, Lars Grunske, Matthias Tichy. "Co-Evolution of Software Architecture and Fault Tree models: An Explorative Case Study on a Pick and Place Factory Automation System." In: *Intl. Workshop on NIM-ALP*. 2013, pp. 32–40.

[60] Sinem Getir, Esteban Pavese, Lars Grunske. "Formal Semantics for Probabilistic Verification of Stochastic Regular Expressions." In: *Proceedings of the 27th International Workshop on Concurrency, Specification and Programming, Berlin, Germany, September 24-26, 2018*. 2018.

[61] Sinem Getir, Michaela Rindt, Timo Kehrer. "A Generic Framework for Analyzing Model Co-Evolution." In: *Proceedings of the Workshop on Models and Evolution co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, Sept 28, 2014*. Ed. by Alfonso Pierantonio, Bernhard Schätz, Dalila Tamzalit. Vol. 1331. CEUR Workshop Proceedings. 2014, pp. 12–21.

[62] Sinem Getir, Duc Anh Vu, Francois Peverali, Daniel Strüber, Timo Kehrer. "State Elimination as Model Transformation Problem." In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017.* 2017, pp. 65–73.

[63] Carlo Ghezzi. "Evolution, Adaptation, and the Quest for Incrementality." In: *Monterey Workshop*. 2012, pp. 369–379.

[64] Carlo Ghezzi. "Of software and change." In: *Journal of Software: Evolution and Process* 29.9 (2017). e1888 smr.1888, e1888. DOI: 10.1002/smr.1888.

[65] Dora Giammarresi, Jean-Luc Ponty, Derick Wood. "The Glushkov and Thompson constructions: A synthesis." In: *Unpublished manuscript, July* 186 (1998).

[66] Marcelo Glusman, Shmuel Katz. "Model Checking Conformance with Scenario-Based Specifications." In: *Computer Aided Verification*. Ed. by Warren A. Hunt, Fabio Somenzi. Springer Berlin Heidelberg, 2003, pp. 328–340.

[67] *Google Cluster Data*. https://github.com/google/cluster-data. [Online; last access December 2020].

[68] Katerina Goseva-Popstojanova, Kishor S. Trivedi. "Architecture-based approach to reliability assessment of software systems." In: *Perform. Eval.* 45.2-3 (2001), pp. 179–204.

[69] Hermann Gruber, Jan Johannsen. "Optimal Lower Bounds on Regular Expression Size Using Communication Complexity." In: *Foundations of Software Science and Computational Structures*. Ed. by Roberto Amadio. Springer Berlin Heidelberg, 2008, pp. 273–286. ISBN: 978-3-540-78499-9.

[70] Dick Grune, Ceriel J. H. Jacobs. *Parsing Techniques - A Practical Guide*. Monographs in Computer Science. Springer, 2008. ISBN: 978-0-387-20248-8. DOI: 10.1007/978-0-387-68954-8.

[71] Lars Grunske. "Specification patterns for probabilistic quality properties." In: *30th International Conference on Software Engineering (ICSE 2008)*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, Volker Gruhn. ACM, 2008, pp. 31–40. DOI: 10.1145/1368088.1368094.

[72] Lars Grunske, Jun Han. "A Comparative Study into Architecture-Based Safety Evaluation Methodologies Using AADL's Error Annex and Failure Propagation Models." In: *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008*. IEEE Computer Society, 2008, pp. 283–292. DOI: 10.1109/HASE.2008.32.

[73] Matthias Güdemann, Frank Ortmeier. "Probabilistic Model-Based Safety Analysis." In: *Proceedings Eighth Workshop on Quantitative Aspects of Programming Languages*. Ed. by Alessandra Di Pierro, Gethin Norman. Vol. 28. EPTCS. 2010, pp. 114–128.

[74] Christian W. Guenther, Stefanie Rinderle-Ma, Manfred Reichert, Wil M.P. van der Aalst, Jan Recker. "Using Process Mining to Learn from Process Changes in Evolutionary Systems." In: *Int'l Journal of Business Process Integration and Management, Special Issue on Business Process Flexibility* 3.1 (2008), pp. 61–78.

[75] Serge Haddad, Patrice Moreaux. "Stochastic Petri Nets." In: 2010, pp. 269 –302. ISBN: 9780470611647. DOI: 10.1002/9780470611647.ch9.

[76] Ernst Moritz Hahn, Holger Hermanns, Lijun Zhang. "Probabilistic Reachability for Parametric Markov Models." In: *Proceedings of the 16th Int. SPIN Workshop on Model Checking Software*. Springer-Verlag, 2009, pp. 88–106. ISBN: 978-3-642-02651-5. DOI: 10.1007/978-3-642-02652-2_10.

[77] Ernst Moritz Hahn, Holger Hermanns, Lijun Zhang. "Probabilistic reachability for parametric Markov models." In: *STTT* 13.1 (2011), pp. 3–19.

[78] Tingting Han, Joost-Pieter Katoen, Berteun Dammans. "Counterexample Generation in Probabilistic Model Checking." In: *IEEE Trans. Software Eng.* 35.2 (2009), pp. 241–257.

[79] Hans Hansson, Bengt Jonsson. "A Logic for Reasoning about Time and Reliability." In: *Formal Aspects of Computing* 6 (1994), pp. 102–111.

[80] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Marco A. A. Sanvido. "Extreme Model Checking." In: *Verification: Theory and Practice*. 2003, pp. 332–358.

[81] Holger Hermanns, Jan Krcál, Jan Kretínský. "Compositional Verification and Optimization of Interactive Markov Chains." In: *CONCUR*. 2013, pp. 364–379.

[82] Holger Hermanns, Jan Krčál, Jan" Křetínský. "Probabilistic Bisimulation: Naturally on Distributions." In: *Concurrency Theory (CONCUR)*. Ed. by Paolo Baldan, Daniele Gorla. Springer Berlin Heidelberg, 2014, pp. 249–265.

[83] Holger Hermanns, Björn Wachter, Lijun Zhang. "Probabilistic CEGAR." In: *Computer Aided Verification*. 2008, pp. 162–175.

[84] Georg Hinkel. "An NMF Solution to the State Elimination Case at the TTC 2017." In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*. Ed. by Antonio García-Domínguez, Georg Hinkel, Filip Krikava. Vol. 2026. CEUR Workshop Proceedings. 2017, pp. 75–79.

[85] Tony Hoare, Bernhard MÃ¶ller, Georg Struth, Ian Wehrman. "Concurrent Kleene Algebra and its Foundations." In: *The Journal of Logic and Algebraic Programming* 80.6 (2011). Relations and Kleene Algebras in Computer Science, pp. 266 –296. ISSN: 1567-8326. DOI: \url{https://doi.org/10.1016/j.jlap.2011.04.005}.

[86] Chih-Duo Hong, Anthony W. Lin, Rupak Majumdar, Philipp Rümmer. "Probabilistic Bisimulation for Parameterized Systems." In: *Computer Aided Verification*. Ed. by Isil Dillig, Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 455–474.

[87] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321455363.

[88] *Incremental Verification with SRE model checking*. https://github.com/sinemgetir/iqon.

[89] Kenneth Johnson, Radu Calinescu, Shinji Kikuchi. "An incremental verification framework for component-based software systems." In: *CBSE'13, Proceedings of the 16th ACM Symposium on Component Based Software Engineering, part of Comparch '13*. 2013, pp. 33–42. DOI: 10.1145/2465449.2465456.

[90] Alexander Kartzow, Thomas Weidner. "Model Checking Constraint LTL over Trees." In: *CoRR* abs/1504.06105 (2015).

[91] Joost-Pieter Katoen. "The Probabilistic Model Checking Landscape." In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '16. New York, NY, USA: ACM, 2016, pp. 31–45. ISBN: 978-1-4503-4391-6. DOI: 10.1145/2933575.2934574.

[92] Mark Kattenbelt, Marta Z. Kwiatkowska, Gethin Norman, David Parker. "A game-based abstraction-refinement framework for Markov decision processes." In: *Formal Methods in System Design* 36.3 (2010), pp. 246–280.

[93] Timo Kehrer. "Calculation and propagation of model changes based on user-level edit operations: a foundation for version and variant management in model-driven engineering." PhD thesis. University of Siegen, 2015.

[94] Donald E. Knuth. "Semantics of Context-Free Languages." In: *Math. Syst. Theory* 2.2 (1968), pp. 127–145. DOI: 10.1007/BF01692511.

[95] Maximilian Koegel, Markus Herrmannsdoerfer, Yang Li, Jonas Helming, Joern David. "Comparing State- and Operation-Based Change Tracking on Models." In: *2010 14th IEEE International Enterprise Distributed Object Computing Conference*. 2010, pp. 163–172.

[96] Dexter C. Kozen. "Kleene Algebra and Regular Expressions." In: *Automata and Computability*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1977, pp. 55–60. ISBN: 978-3-642-85706-5. DOI: 10.1007/978-3-642-85706-5_10.

[97] Dexter C. Kozen. "A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events." In: *Inf. Comput.* 110.2 (1994), pp. 366–390. DOI: 10.1006/inco.1994.1037.

[98] Jan Kretinsky. *Quantitative Verification*. https://www7.in.tum.de/um/courses/QV/ws1819/qv.html. Online; last access December 2020. 2019.

[99] Ratnesh Kumar, Vijay K. Garg. "Control of stochastic discrete event systems modeled by probabilistic languages." In: *IEEE Trans. Automat. Contr.* 46.4 (2001), pp. 593–606. DOI: 10.1109/9.917660.

[100] Marta Z. Kwiatkowska, Gethin Norman, David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems." In: *Computer Aided Verification*. 2011, pp. 585–591.

[101] Marta Z. Kwiatkowska, Gethin Norman, David Parker, Hongyang Qu. "Assume-Guarantee Verification for Probabilistic Systems." In: *TACAS*. 2010, pp. 23–37.

[102] Marta Z. Kwiatkowska, David Parker, Hongyang Qu. "Incremental quantitative verification for Markov decision processes." In: *DSN*. 2011, pp. 359–370.

[103] Marta Kwiatkowska, Gethin Norman, David Parker. "Probabilistic model checking in practice: Case studies with PRISM." In: *ACM SIGMETRICS Performance Evaluation Review* 32.4 (2005), pp. 16–21.

[104] Marta Kwiatkowska, Gethin Norman, David Parker. *Advances and Challenges of Probabilistic Model Checking*. 2010.

[105] Christoph Legat, Jens Folmer, Birgit Vogel-Heuser. "Evolution in industrial plant automation: A case study." In: *39th Annual Conference of the IEEE Industrial Electronics Society*. IEEE Computer Society, pp. 4386–4391.

[106] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." In: Jan. 2013, pp. 1–32. DOI: 10.1007/978-3-642-35813-5_1.

[107] Martin Leucker, César Sánchez. "Regular Linear Temporal Logic." In: *Theoretical Aspects of Computing – ICTAC 2007*. Ed. by Cliff B. Jones, Zhiming Liu, Jim Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 291–305. ISBN: 978-3-540-75292-9.

[108] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[109] Yang Liu, Chao He. "A Heuristics-Based Incremental Probabilistic Model Checking at Runtime." In: *2020 IEEE 11th International Conference on Software Engineering and Service Science (ICSESS)*. 2020, pp. 355–358. DOI: 10.1109/ICSESS49938.2020.9237680.

[110] Yamilet R. Serrano Llerena, Marcel Böhme, Marc Brünink, Guoxin Su, David S. Rosenblum. "Verifying the long-run behavior of probabilistic system models in the presence of uncertainty." In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 2018, pp. 587–597. DOI: 10.1145/3236024.3236078.

[111] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas Nielsen, Kim Larsen, Brian Nielsen. "Learning Probabilistic Automata for Model Checking." In: Oct. 2011, pp. 111 –120. DOI: 10.1109/QEST.2011.21.

[112] M. Ajmone Marsan. "Stochastic Petri nets: An elementary introduction." In: *Advances in Petri Nets 1989", year="1990*. Ed. by Grzegorz Rozenberg. Springer Berlin Heidelberg, pp. 1–29.

[113] Annabelle McIver, Tahiry M. Rabehaja, Georg Struth. "Probabilistic Concurrent Kleene Algebra." In: *Proceedings 11th International Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL 2013, Rome, Italy, March 23-24, 2013*. 2013, pp. 97–115. DOI: 10.4204/EPTCS.117.7.

[114] Kenneth L. McMillan. "Lazy Abstraction with Interpolants." In: *Computer Aided Verification*. 2006, pp. 123–136.

[115] Robert McNaughton, Hisao Yamada. "Regular Expressions and State Graphs for Automata." In: *IRE Trans. Electron. Comput.* 9 (1960), pp. 39–47.

[116] Indika Meedeniya, Lars Grunske. "An Efficient Method for Architecture-Based Reliability Evaluation for Evolving Systems with Changing Parameters." In: *ISSRE 2010*. IEEE Computer Society, 2010, pp. 229–238.

[117] Benjamin Monmege. "Specification and verification of quantitative properties : expressions, logics, and automata." Theses. École normale supérieure de Cachan - ENS Cachan, Oct. 2013.

[118] Nelma Moreira, Davide Nabais, RogÃ©rio Reis. "State Elimination Ordering Strategies: Some Experimental Results." In: *Electronic Proceedings in Theoretical Computer Science* 31 (2010), 139â€"148. ISSN: 2075-2180. DOI: 10.4204/eptcs.31.16.

[119] Christoph Neumann. "Converting deterministic finite automata to regular expressions." In: (2005). [Online; last access December 2020].

[120] Gethin Norman, David Parker. *Quantitative Verification: Formal Guarantees for Timeliness, Reliability and Performance*. Technical Report. London Mathematical Society and Smith Institute, 2014.

[121] Yiannis Papadopoulos, Christian Grante. "Evolving car designs using model-based automated safety analysis and optimisation techniques." In: *The Journal of Systems and Software* 76.1 (2005), pp. 77–89.

[122] Yiannis Papadopoulos, John A. McDermid, Ralph Sasse, Gunter Heiner: "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure." In: *Int. Journal of Reliability Engineering and System Safety* 71.3 (2001), pp. 229–247.

[123] David Parker. *PCTL model checking for DTMCs*. `https://www.prismmodelchecker.org/lectures/pmc/05-dtmcmodelchecking.pdf`. Online; last access December 2020.

[124] M. Procopiuc, O. Procopiuc, S. H. Rodger. "Visualization and interaction in the computer science formal languages course with JFLAP." In: *Technology-Based Re-Engineering Engineering Education Proceedings of Frontiers in Education FIE'96 26th Annual Conference*. Vol. 1. 1996, 121–125 vol.1.

[125] Michael O. Rabin. "Probabilistic Automata." In: *Information and Control* 6.3 (1963), pp. 230–245. DOI: `10.1016/S0019-9958(63)90290-0`.

[126] Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, Lukas Märtin. *Managed Software Evolution*. Springer, Cham, June 2019. ISBN: 9783030134990. DOI: `10.1007/978-3-030-13499-0`.

[127] Genaína Nunes Rodrigues, David S. Rosenblum, Sebastián Uchitel. "Using Scenarios to Predict the Reliability of Concurrent Component-Based Software Systems." In: *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005*. Ed. by Maura Cerioli. Vol. 3442. Lecture Notes in Computer Science. Springer, 2005, pp. 111–126. ISBN: 3-540-25420-X.

[128] Brian J. Ross. "Probabilistic Pattern Matching and the Evolution of Stochastic Regular Expressions." In: *Applied Intelligence* 13.3 (Nov. 2000), pp. 285–300. ISSN: 0924-669X.

[129] Anne Rozinat, Wil van der Aalst. *Conformance testing : measuring the alignment between event logs and process models*. English. BETA publicatie : working papers. Technische Universiteit Eindhoven, 2005. ISBN: 90-386-0077-1.

[130] Peter Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, Anthony Finkelstein. "Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems." In: *RE 2010, 18th IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 95–103.

[131] Mohammadreza Sharbaf, Shekoufeh Kolahdouz Rahimi, Bahman Zamani. "Solving the State Elimination Case Study Using Epsilon." In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*. Ed. by Antonio García-Domínguez, Georg Hinkel, Filip Krikava. Vol. 2026. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 91–95.

[132] Andrew Stevenson, James R. Cordy. "A survey of grammatical inference in software engineering." In: *Science of Computer Programming* 96 (2014). Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012), pp. 444 –459. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2014.05.008.

[133] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1981. ISBN: 0262690764.

[134] Daniel Strüber. "Transformation of Finite State Automata to Regular Expressions Using Henshin." In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*. Ed. by Antonio García-Domínguez, Georg Hinkel, Filip Krikava. Vol. 2026. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 81–85.

[135] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, Matthias Tichy. "Henshin: A Usability-Focused Framework for EMF Model Transformation Development." In: *Graph Transformation*. Ed. by Juan de Lara, Detlef Plump. Cham: Springer International Publishing, 2017, pp. 196–208.

[136] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms." In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.

[137] Alex Thomo. "Implication of regular expressions." In: *Appl. Math. Lett.* 25.10 (2012), pp. 1394–1398. DOI: 10.1016/j.aml.2011.12.009.

[138] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm." In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387.

[139] Mirco Tribastone, Stephen Gilmore. "Automatic extraction of PEPA performance models from UML activity diagrams annotated with the MARTE profile." In: *WOSP*. 2008, pp. 67–78.

[140] Alphan Ulusoy, Tichakorn Wongpiromsarn, Calin Belta. "Incremental control synthesis in probabilistic environments with Temporal Logic constraints." In: *Proceedings of the 51th IEEE Conference on Decision and Control, CDC 2012, December 10-13, 2012, Maui, HI, USA*. 2012, pp. 7658–7663.

[141] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, John Wilkes. "Large-scale cluster management at Google with Borg." In: *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France, 2015.

[142] Thomas Weidner. "Probabilistic Logic, Probabilistic Regular Expressions, and Constraint Temporal Logic." PhD dissertation. University Leipzig, 2012.

[143] Thomas Weidner. "Probabilistic Regular Expressions and MSO Logic on Finite Trees." In: *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*. Ed. by Prahladh Harsha, G. Ramalingam. Vol. 45. Leibniz International Proceedings in Informatics (LIPIcs). 2015, pp. 503–516. ISBN: 978-3-939897-97-2.

[144] Alexander Weidt, Albert Zündorf. "The SDMLib Solution to the TTC 2017 State Elimination Case." In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017*. Ed. by Antonio García-Domínguez, Georg Hinkel, Filip Krikava. Vol. 2026. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 87–89.

[145] Danny Weyns et al. "Perpetual Assurances for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems III. Assurances - International Seminar, Dagstuhl Castle, Germany, 2013, Revised Selected and Invited Papers*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, Holger Giese. Vol. 9640. Lecture Notes in Computer Science. Springer, 2013, pp. 31–63. DOI: 10.1007/978-3-319-74183-3_2.

[146] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, Jean-Michel Bruel. "RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems." In: *RE 2009, 17th IEEE International Requirements Eng. Conference*. IEEE Computer Society, 2009, pp. 79–88.

[147] C. Murray Woodside, Dorina C. Petriu, Dorin Bogdan Petriu, Hui Shen, Toqeer Israr, José Merseguer. "Performance by unified model analysis (PUMA)." In: *Proceedings of the Fifth International Workshop on Software and Performance, WOSP 2005*. ACM, 2005, pp. 1–12.

[148] Xingyu Zhao, Valentin Robu, David Flynn, Fateme Dinmohammadi, Michael Fisher, Matt Webster. "Probabilistic Model Checking of Robots Deployed in Extreme Environments." In: *The Thirty-Third AAAI Conference on Artificial Intelligence AAAI, The Ninth Symposium on Educational Advances in Artificial Intelligence EAAI*. AAAI Press, 2019, pp. 8066–8074. DOI: 10.1609/aaai.v33i01.33018066. URL: https://doi.org/10.1609/aaai.v33i01.33018066.

[149] Tao Zheng, C. Murray Woodside, Marin Litoiu. "Performance Model Estimation and Tracking Using Optimal Filters." In: *IEEE Trans. Software Eng* 34.3 (2008), pp. 391–406.

PUBLICATIONS

1. *Sinem Getir, Esteban Pavese, Lars Grunske,*
   "Quantitative Verification of Stochastic Regular Expressions" In: Fundamenta Informaticae, 2020 (in press).

2. *Sinem Getir, Lars Grunske, André van Hoorn, Timo Kehrer, Yannic Noller, Matthias Tichy,*
   "Supporting semi-automatic co-evolution of architecture and fault tree models" In: Journal of Systems and Software 142, pp. 115–135, 2018.

3. *Sinem Getir, Esteban Pavese, Lars Grunske,*
   "Formal Semantics for Probabilistic Verification of Stochastic Regular Expressions" In: Proceedings of the 27th International Workshop on Concurrency, Specification and Programming, 2018.

4. *Stefan Kögel, Matthias Tichy, Abhishek Chakraborty, Alexander Fay, Birgit Vogel-Heuser, Christopher Haubeck, Gabriele Taentzer, Timo Kehrer, Jan Ladiges, Lars Grunske, Mattias Ulbrich, Safa Bougouffa, Sinem Getir, Suhyun Cha, Udo Kelter, Winfried Lamersdorf, Kiana Busch, Robert Heinrich, Sandro Koch,*
   "Learning from Evolution for Evolution", Managed Software Evolution, pp. 255-308, 2019.

5. *Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, Jürgen Walter,*
   "Performance Analysis Strategies for Software Variants and Versions" Managed Software Evolution, pp. 175-206, 2019.

6. *Sinem Getir, Duc Anh Vu, Francois Peverali, Daniel Strüber, Timo Kehrer,*
   "State Elimination as Model Transformation Problem" In: Proceedings of the 10th Transformation Tool Contest (TTC), co-located with the Software Technologies: Applications and Foundations (STAF), pp. 65–73, 2017.

7. *Sinem Getir, Lars Grunske, Christian Karl Bernasko, Verena Käfer, Tim Sanwald, Matthias Tichy,*
   "CoWolf - A Generic Framework for Multi-view Co-evolution and Evaluation of Models" ICMT, pp. 34-40, 2015.

8. *Birgit Vogel-Heuser, Stefan Feldmann, Jens Folmer, Jan Ladiges, Alexander Fay, Sascha Lity, Matthias Tichy, Matthias Kowal, Ina Schaefer, Christopher Haubeck, Winfried Lamersdorf, Timo Kehrer, Sinem Getir, Mattias Ulbrich, Vladimir Klebanov, Bernhard Beckert,*
   "Selected challenges of software evolution for automated production systems" INDIN, pp. 314-321, 2015.

9. *Sinem Getir, Micheala Rindt and Timo Kehrer,*
   "A Generic Framework for Analyzing Model Co-Evolution" In: Model Evolution, MODELS, 2014.

10. *Sinem Getir, Lars Grunske, Matthias Tichy*,
ENSURE: Ensurance of Software Evolution by Run-time Certification-Description of Selected Project Results, Tagungsband des Dagstuhl-Workshops, pp. 23, 2014.

11. *Sinem Getir, André Van Hoorn, Lars Grunske, Matthias Tichy*,
"Co-Evolution of Software Architecture and Fault Tree models: An Explorative Case Study on a Pick and Place Factory Automation System", In: International Workshop Non-functional Properties in Modeling: Analysis, Languages, and Processes, MODELS, pp. 32-49, 2013.

DECLARATION

Hiermit erkläre ich, dass ich die beigefügte Dissertation mit dem Titel "Quantitative Modeling and
Verification of Evolving Software" selbstständig angefertigt und verfasst habe, dass ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe und dass alle Stellen, die wörtlich oder dem Sinne nach aus anderen Veröffentlichungen und Quellen entnommen wurden, kenntlich gemacht sind. Ich versichere außerdem, dass die beigefügte Dissertation und Arbeit bisher an keiner anderen Hochschule eingereicht wurde.

Berlin, December 2020

Sinem Getir Yaman

I declare that this thesis entitled "Quantitative Modeling and
Verification of Evolving Software" has been composed solely by myself, that I have not used any other sources and resources than the ones referenced in this thesis, and that I have marked all material in this thesis with regards to quotations and content by referencing the corresponding publications or sources. Furthermore, I declare that this thesis has not been submitted, in whole or in part, in any previous application for a degree to another university.

Berlin, December 2020

Sinem Getir Yaman