

Imperial College London
Department of Computing

Compositional Behaviour and Reliability

Models for Adaptive Component-based Architectures

Pedro Miguel Fonseca Rodrigues

August 2015

Supervised by Emil C. Lupu and Jeff Kramer

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London



Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Copyright

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

Abstract

The increasing scale and distribution of modern pervasive computing and service-based platforms makes manual maintenance and evolution difficult and too slow. Systems should therefore be designed to self-adapt in response to environment changes, which requires the use of on-line models and analysis. Although there has been a considerable amount of work on architectural modelling and behavioural analysis of component-based systems, there is a need for approaches that integrate the architectural, behavioural and management aspects of a system. In particular, the lack of support for composability in probabilistic behavioural models prevents their systematic use for adapting systems based on changes in their non-functional properties. Of these non-functional properties, this thesis focuses on reliability.

We introduce Probabilistic Component Automata (PCA) for describing the probabilistic behaviour of those systems. Our formalism simultaneously overcomes three of the main limitations of existing work: it preserves a close correspondence between the behavioural and architectural views of a system in both abstractions and semantics; it is composable as behavioural models of composite components are automatically obtained by combining the models of their constituent parts; and lastly it is probabilistic thereby enabling analysis of non-functional properties. PCA also provide constructs for representing failure, failure propagation and failure handling in component-based systems in a manner that closely corresponds to the use of exceptions in programming languages. Although PCA is used throughout this thesis for reliability analysis, the model can also be seen as an abstract process algebra that may be applicable for analysis of other system properties.

We further show how reliability analysis based on PCA models can be used to perform architectural adaptation on distributed component-based systems and evaluate the computational cost of decentralised adaptation decisions. To mitigate the state-explosion problem associated with composite models, we further introduce an algorithm to reduce a component's PCA model to one that only represents its interface behaviour. We formally show that such model preserves the properties of the original representation. By experiment, we show that the reduced models are significantly smaller than the original, achieving a reduction of more than 80% on both the number of states and transitions. A further benefit of the approach is that it allows component profiling and probabilistic interface behaviour to be extracted independently for each component, thereby enabling its exchange between different organisations without revealing commercially sensitive aspects of the components' implementations.

The contributions and results of this work are evaluated both through a series of small scale examples and through a larger case study of an e-Banking application derived from Java EE training materials. Our work shows how probabilistic non-functional properties can be integrated with the architectural and behavioural models of a system in an intuitive and scalable way that enables automated architecture reconfiguration based on reliability properties using composable models.

Acknowledgements

I would like to first express my deep gratitude to my supervisors, Emil Lupu and Jeff Kramer, for their generous support, guidance, encouragement and wisdom which have been invaluable over the course of my PhD. I am forever grateful and thankful to them for helping me to grow as a researcher. I would also like to thank my thesis examiners, Anthony Finkelstein and Michael Huth, for their time, comments and examination of the work.

I would like to acknowledge the funding I received from Fundação para a Ciência e Tecnologia (grant number SFRH/BD/73967/2010), without which this work could not have been conducted.

I also would like to thank everyone that I had the pleasure to share an office with, who made it a more pleasant environment: Srdjan, Rudi, Anandha, Edmund, Jeremiah, Luke, Jiefei, Graham, Anna, Mark and Jim. My thanks is also extended to the members of the ERATS group: Daniele, Vittorio, Luis, Martin and Federico.

A very special thanks goes to my family. I am especially indebted to my father Joaquin, my mother Emília and my sister Ana for their unconditional support. I would not be here if it were not for their help and guidance.

Finally, I am forever grateful to my future wife Catarina, who has always been by my side, specially during this long four year journey.

Contents

Abstract	3
Acknowledgements	7
1. Introduction	21
1.1. Requirements	24
1.1.1. R1 - Autonomy	25
1.1.2. R2 - Integrated System Models	25
1.1.3. R3 - Composability	25
1.1.4. R4 - Scalability	26
1.1.5. R5 - Adaptation Safety	26
1.1.6. R6 - Decentralisation	26
1.2. Contributions	27
1.3. Thesis Outline	28
2. Background	31
2.1. Adaptation Techniques	31
2.1.1. Programming Languages Support	32
2.1.2. Interception as a Means to Implement Adaptation	33
2.1.3. Proxies to Implement Adaptation	34
2.2. Software Models	34
2.2.1. Architectural models	34
2.2.2. Behaviour Models	38
2.3. Management	42
2.3.1. Decision-Making	42
2.3.2. Languages for Action Policies	43
2.3.3. Using Utility-functions	45

2.3.4. Mechanisms Supporting Goal Policies	46
2.3.5. Self-Managing Frameworks	46
2.4. Adaptation Correctness	50
2.5. Integrated Models	53
2.5.1. Architecture and Behaviour models	53
2.5.2. Architecture and Adaptation	55
2.5.3. Behaviour and Adaptation	56
2.6. Background Summary	59
3. Modelling Probabilistic Behaviour and Reliability	63
3.1. Comparison between CTMC and DTMC models	64
3.2. Models of probabilistic choice	68
3.2.1. Composable Probabilistic Models	72
3.3. Probabilistic Component Automata	81
3.3.1. Modelling Basic Components	84
3.3.2. Modelling Composite Components	86
3.3.3. Removing Unused Behaviour	90
3.3.4. Relabelling	93
3.4. Reliability Modelling and Analysis	98
3.5. Composite Analysis	100
3.5.1. Parallel Composition Algorithm	100
3.5.2. Reliability Analysis	103
3.5.3. Hiding	104
3.5.4. Profiling	104
3.6. Tool Support - LTSA-PCA	105
3.7. Summary and Related Work	106
4. Towards Scalable Compositional Reliability Analysis	109
4.1. Reduction Algorithm	110
4.1.1. Algorithm Description	114
4.1.2. Examples	119
4.2. Evaluation	128
4.2.1. Time Complexity	128

4.2.2. Empirical Evaluation	131
4.3. Formal Verification	133
4.4. Summary and Related Work	135
5. Automating Architectural Reconfiguration	137
5.1. Notational Elements	138
5.1.1. Architectural View	139
5.1.2. Behavioural View	140
5.1.3. Management View	141
5.2. Architectural reconfiguration	142
5.2.1. Centralised Algorithm	143
5.2.2. Example Web System for Centralised Reconfiguration	145
5.2.3. Distributed Reconfiguration	147
5.2.4. Example WebSystem for Distributed Reconfiguration Algorithm	151
5.3. Evaluation	154
5.4. Summary	155
6. Evaluation	159
6.1. e-Banking Case Study System	159
6.1.1. Modelling Probabilistic Behaviour	160
6.1.2. Composite Model	165
6.2. Probabilistic Analysis	169
6.2.1. Sensitivity Analysis	172
6.3. Summary	174
6.4. Qualitative Evaluation	175
7. Conclusions	177
7.1. Contributions	178
7.2. Future Work	180
Bibliography	183
Appendix	195

A. Notions of Equivalence for Probabilistic Component Automata	195
A.1. Behaviour Equivalences for Labelled Transition Systems	196
A.1.1. Strong Bisimulation	197
A.1.2. Weak Bisimulation	197
A.2. Behavioural Equivalences for Probabilistic Labelled Transitions Systems	198
A.2.1. Strong Bisimulation	199
A.2.2. Weak Bisimulation	203
A.3. Behaviour Equivalences for Probabilistic Component Automata	207
A.3.1. Strong Bisimulation	209
A.3.2. Weak Bisimulation	211
A.4. Hiding Operator	215
A.4.1. Preliminaries	215
A.4.2. Transformation T_1	215
A.4.3. Transformation T_2	226
A.4.4. Transitivity	234
A.5. Congruence	239
A.5.1. Hiding	241
A.5.2. Interface	241
A.5.3. Relabelling	242
A.5.4. Parallel Composition	243
A.6. Summary	250
B. Tool Support - LTSA-PCA	253

List of Tables

4.1. Reduction Algorithm Evaluation Results - Web-Server Systems 1 and 2	133
5.1. Architectural Assembly Evaluation Results	154
6.1. Size of PCA Models for Composition	168
6.2. Size of Original and Reduced Composite Models for the e-Banking System	169
6.3. Reliability Properties for the e-Banking System	170
6.4. Time to Analyse Reliability and Operational Properties for the e-Banking System	171

List of Figures

2.1. Example Architecture of Video Game using C2 ADL [Med96]	36
3.1. Activity Diagram of an e-commerce System [FGT12]	65
3.2. DTMC Model of e-commerce System [FGT12]	65
3.3. CTMC Model of e-commerce System [FGT12]	67
3.4. Examples of DTMC Models	69
3.5. Synchronous Composition of DTMC Models A and B	69
3.6. Composition of DTMC Models A and B	70
3.7. Examples of Generative PLTS Models	73
3.8. Synchronous Composition of Generative PLTS Models A and B	74
3.9. Generative PLTS Model C	74
3.10. Synchronous Composition of Generative PLTS Models A and C	75
3.11. Darwin Representations of Client Component	83
3.12. Darwin Representations of Server Component	83
3.13. Client Component and Associated PCA Representation	83
3.14. Server Component and Associated PCA Representation	84
3.15. Example Case for Choice Operator with Input Actions	85
3.16. Client-Server System Models	88
3.17. Composite PCA for Client-Server System Using Normalisation Defined by η for PCA Models	88
3.18. Composite PCA for Client-Server System Using Normalisation Defined by σ [BBS95]	89
3.19. Composite PCA for Client-Server System Using Normalisation Defined by PIOA	90
3.20. Client-Server System Architecture with Unbound Provided Interface	91
3.21. Server PCA with Two Provided Interfaces	91
3.22. Client-Server Composite PCA with Unused Behaviour	92
3.23. Server PCA with Two Provided Interfaces - Shared Behaviour	92

3.24. Two Clients-Server Architectural Configuration	94
3.25. Server PCA with Multiple Bindings	95
3.26. Client 1 and 2 PCA Models	95
3.27. Client-Server Composite PCA with Multiple Bindings	97
3.28. Server PCA with Failures	98
3.29. Client-Server Composite PCA with Failures	99
3.30. Client PCA with Failure Handling	99
3.31. Client-Server Composite PCA with Failure Handling	100
4.1. Forward Propagation of a Single Transition	112
4.2. Forward Propagation of Multiple Transitions	112
4.3. Backward Propagation of Multiple Transitions	113
4.4. Example PCA Model for Reduction Algorithm	119
4.5. Example PCA Model for Reduction Algorithm - Output of First Phase	120
4.6. Example PCA Model for Reduction Algorithm - Reduced Model	121
4.7. Example PCA Model A_1 for Forward Propagation	121
4.8. Example PCA Model A_2 for Forward Propagation - Input Actions	122
4.9. Example Reduced PCA Models from A_2 for Forward Propagation - Input Actions	122
4.10. Example PCA model B - Preservation of Compositional Semantics	123
4.11. Example Composite Models for Preservation of Compositional Semantics	123
4.12. Example PCA Models C and D for Preservation of Compositional Semantics	123
4.13. Composite PCA Model $C\ D$ - Preservation of Compositional Semantics	124
4.14. Reduced PCA Models C' and Composite Model $C'\ D$ for Preservation of Compositional Semantics	125
4.15. Example PCA Model G - Propagation of Multiple Transitions from Multiple Source States	125
4.16. Example Reduced PCA Model G' - Propagation of Multiple Transitions from Multiple Source States	126
4.17. Example Reduced PCA Model H - Invalid Propagation of Multiple Transitions	126
4.18. Example PCA Model I - Reduction of Cyclic Paths	127
4.19. Reduced PCA Model I' after Executing First Phase on Model I - Reduction of Cyclic Paths	127
4.20. Final Reduced PCA Model I'_2 from PCA I - Reduction of Cyclic Paths	128

4.21. Web-Server Architectural Configuration	131
4.22. Web-Server2 Architectural Configuration	132
5.1. Darwin Representations of Client Component	139
5.2. Darwin Representations of Server Component	139
5.3. Darwin Representations of Client-Server System	140
5.4. Description of Configuration $\mathcal{B}_1$ of the WebSystem using Darwin	146
5.5. Description of Configuration $\mathcal{B}_2$ of the WebSystem using Darwin	147
5.6. Configuration Used to Generate Intermediate PCA for Server Component	152
5.7. Configuration of Backend-1 Component in System Configuration $\mathcal{B}_2$	154
6.1. Architectural Configuration of the e-Banking System from [Ora]	160
6.2. Partial PCA Representation of Account Java Persistent Entity	161
6.3. PCA representation for createAccount method of AccountControllerBean	164
6.4. Composite Account Controller Bean	166
6.5. Composite Web Client	167
6.6. Composite App Client	167
6.7. Composite e-Banking System	169
6.8. Sensitivity Analysis on the e-Banking System	173
6.9. Sensitivity Analysis on the AccountControllerBean Actions Using PCA and DTMC	174
A.1. Examples of PLTS Models for Strong Bisimulation	200
A.2. Disjoint Union PLTS $\hat{A}$ for Strong Bisimulation Between PLTS Models A_1 and A_2	201
A.3. Examples of PLTS Models for Weak Bisimulation	204
A.4. Disjoint Union PLTS $\hat{A}$ for Weak Bisimulation Between PLTS Models A_1 and A_2	205
A.5. Examples of PCA Models for Strong Bisimulation	209
A.6. Disjoint Union PCA $\hat{A}$ for Strong Bisimulation Between PCA Models A_1 and A_2	210
A.7. Examples of PCA Models for Weak Bisimulation	212
A.8. Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2	213
A.9. Examples of PCA Models for Weak Bisimulation - Transformation T_1	223
A.10. Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_1	225

A.11.Examples of PCA Models for Weak Bisimulation - Transformation T_2 : Same Source State	228
A.12.Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_2 : Same Source State	229
A.13.Examples of PCA Models for Weak Bisimulation - Transformation T_2 : Multiple Source States	232
A.14.Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_1	234
A.15.Examples of PCA Models for Transitivity Property	240
A.16.Disjoint Union PCA $\hat{A}$ for Transitivity Property	241
B.1. Architecture of e-commerce System [FGT12]	253
B.2. Original DTMC Model of e-commerce System [FGT12]	254
B.3. Editor Panel - LTSA-PCA	254
B.4. Editor Panel - Authentication Service P-FSP	255
B.5. Editor Panel - Authentication Service PCA	255
B.6. Editor Panel - Shipping Service P-FSP	256
B.7. Transitions Panel - LTSA-PCA	258
B.8. Alphabet Panel - LTSA-PCA	259
B.9. PRISM Panel - LTSA-PCA	260

List of Algorithms

3.1. Interface Operator Algorithm	93
4.1. PCA Reduction Algorithm	110
4.2. First Phase	116
4.3. Second phase	117
4.4. collapseForward(s, s', μ_{cf})	119
4.5. collapseBackward(s, s', μ_{cb})	119
5.1. Distributed Assembly Algorithm	148

1. Introduction

Due to their increasing complexity, modern distributed applications need to be permanently maintained and frequently changed to ensure their robustness in the face of a changing environment [ST09, OMT08, CLG⁺09]. Making such changes manually is too difficult, slow and error prone. Systems are therefore required to adapt autonomously in face of changes in their environment or in their operational conditions. On top of that, the rapid pace of change in the execution environment makes it infeasible to anticipate all the possible changes at design-time. To cope with unforeseen situations at runtime, systems should avoid depending only on assumptions made at design w.r.t. the execution environment.

To achieve self-management, systems can resort to different mechanisms to adapt their behaviour and/or structure. These changes can be realised by changing parameters or by introducing proxy components that encapsulate and mediate interactions with other components [SMCS04, MG05]. Transparent methods that do not require changing the source code of the application can also be applied using middleware mechanisms that intercept messages/invocations between components in order to adapt the system's behaviour [SM04, OB99, RC02].

Although these approaches allow to manipulate the behaviour of an application, they tend to be too low-level and specific to a particular programming language and/or application domains. Architectural adaptation, where the system can be reconfigured by changing the bindings between components, provides a suitable level of abstraction for self-managed systems [SHMK08]. In the context of software architectures, a *component* provides a separation between structure and implementation, and supports encapsulation of behaviour. It can equally well represent software libraries, hardware devices or (web) services. Following component-based design, systems are then built by composing loosely coupled components that provide higher-level functionality by combining the capabilities of a set of (sub-)components.

When the system is being assembled or when reconfiguration is required, the set of possible configurations for a system can be constructed from a description of functional requirements and a list of available components. If multiple components offer the same functionality, non

functional properties/Quality-of-Service (QoS) parameters, such as reliability and performance, are often used to choose between alternative configurations.

Monitoring QoS metrics has been extensively studied in Network Management Systems [ACH98]. However, the QoS monitoring in existing management frameworks is usually deployed externally to a running software system [LDS⁺08] and is not aware of the system model, *e.g.* its internal structure. Monitoring non-functional parameters is thus performed for the system as a whole. However, changes in non-functional parameters may be associated with a particular part (component) of the system, or even a particular aspect of the implementation of a single component. *On top of that, non-functional properties that a component provides are also dependent on the context in which the component is deployed.* Although the reliability/performance of the instructions executed by a component plays an important role in determining the reliability/performance provided by a component, the interactions with other components can also have a significant impact. For example, a component may have some functionality that is unreliable or slow, but if it is seldom used then its impact on the overall system reliability/performance is minimal.

Model-based *reliability* analysis of a particular configuration typically analysed by modelling the control flow between components and representing the failure of each component using a Discrete-Time Markov Chain (DTMC) [Che80]. Reliability requirements are defined using Probabilistic Computation Tree Logic (PCTL) [BK08]. The *performance* of the system can be analogously analysed using a Continuous-Time Markov Chain model [FGT12], while performance requirements are specified using Continuous Stochastic Logic (CSL) [BK08]. Automatic verification of reliability and performance requirements for DTMC and CTMC models, respectively, is enabled through Model-checking tools such as PRISM [KNP].

Although this approach has been successfully applied to early reliability assessment of alternative architectural assemblies at design-time [IN08], when the system needs to adapt to changes in requirements and in operational conditions at run-time, because new components become available, or others may face degradation of their service or fail all together, only architectural configurations defined at design-time can be considered if DTMC and CTMC models are used. While DTMC and CTMC models support automatic analysis, the DTMC/CTMC model of a system is not automatically constructed from the models of each component. Consequently, a separate DTMC/CTMC model has to be manually defined for each architectural configuration. When new components become available, human intervention is required to define the

DTMC/CTMC model corresponding to each alternative configuration that involves the new components. In fact, automating architecture reconfiguration at run-time presents significant challenges.

An essential aspect to achieve automation is the composability of models, *i.e.* the ability to derive the composite representation of a system configuration, using the model of each (sub-) component. This has been successfully explored for software architectures [GMW00, MDEK95, MT10], where a system can be structured as a composition of components. Such structure is independent from the distributor/developer of each component as well as the programming language used. As a result, the complexity of designing medium to large-scale systems is reduced by supporting incremental elaboration. To this end, composite components provide higher-level functionality whilst hiding their internal structure to other components. Software architectures also enable reuse of previous successful designs, while the same component can be used in different configurations.

Composable behaviour models using representations based on Labelled Transition Systems (LTS) have also been widely used for analysis of the functional behaviour of distributed systems [MK06, CES86, TK93, YY91, LSW95]. Similarly to software architectures models, behaviour models provide generic high-level structures that are not specific to a particular programming language. When used to model component-based systems, the behaviour of each component is depicted as a separate LTS representation, thereby facilitating incremental modelling. The behaviour model of a composite component (application) is then automatically constructed by composing the individual LTS representation of each component. In most cases, the same representation of a component's behaviour can be used in any architecture configuration that includes that component. LTS representations also support encapsulation of internal behaviour by hiding internal actions. Additional operators enable representation of multiple bindings as shared resources [MK06]. However, LTS models do not include probabilistic information and therefore can only be used for behavioural analysis, *e.g.* deadlock and liveness properties.

Composability is less well supported for non-functional properties which require extending behaviour models with stochastic information on the frequency of execution and/or duration of actions. However, composing stochastic information is a challenging task which involves determining a reasonable interpretation of probabilities in an automaton, which is then preserved when composing with other models [SV04]. In the context of performance analysis, the Performance Evaluation Process Algebra (PEPA) [Hil96] provides support for incremental definition

and composition by exploring the properties of exponential distributions. However, these models impose a different semantics based on the duration of actions, thus presenting limited support for compositional analysis, where the internal behaviour of components is deleted before composing with other components, and the associated stochastic information propagated to remaining actions. In addition, these models also have limited representations for failures, their propagation and handling. They support analysis of non-functional requirements associated with questions such as “what is the probability that the system fails within s units of time?” or “what is the average time until the system fails?”. *In contrast, we focus in this thesis on reliability requirements based on the reachability of failure states through discrete-time steps, which correspond to the following questions: “what is the probability that the system fails?” or “what is the probability of failure after action a ?”.*

Self-management thus requires addressing architectural, behavioural and non-functional aspects along with adaptation as first-class concerns. These aspects are interdependent and should be considered together as part of the design process. However, traditional Software Engineering and Systems Management techniques are not well equipped to achieve this, and tend to consider them separately. The goal of this thesis is to define an extended component-based multi-view modelling approach that makes provisions for these aspects thereby leveraging their complementarities. We aim to preserve composability and compositionality as key aspects across all views and elements of the framework to enable distributed adaptation based on local and distributed control, thereby providing better scalability. Non-functional aspects, such as reliability or other quality-of-service parameters usually drive the selection of new architectural configurations. An additional goal of this thesis is to define automatic re-configuration methods which preserve the system’s reliability using composable models for the representation and analysis of probabilistic behaviour.

1.1. Requirements

Having discussed the motivation for self-management and the key problems associated with existing approaches, we briefly discuss the main requirements related to self-managed architectures.

1.1.1. R1 - Autonomy

Self-management requires a system to be able to adapt autonomously, without requiring user input at run-time. When applied to architectural reconfiguration, it implies determining the possible alternative configurations based on the available components, calculating their non-functional properties and choosing the most suitable configuration that satisfies functional and non-functional requirements. To avoid limiting the alternatives for re-configuration, the system designer should follow a declarative approach, *i.e.* by defining the conditions that adaptation processes need to achieve, instead of specifying the steps for adaptation. The next Sections enumerate the requirements for automated analysis that enable self-management for component-based architectures.

1.1.2. R2 - Integrated System Models

Separation of concerns is a principle widely applied in the development of complex systems, whereby the system functionality is decomposed into smaller units in order to reduce the implementation complexity and/or to focus on a particular concern. The same principle is applied to system models to enable incremental modelling and to focus on a particular aspect of a system. However, self-management requires combining the architectural, behavioural, non-functional properties and management concerns. For instance, the reliability of a component depends not only on its behaviour, but also on how it is used and on the reliability of the components it requires. As a result, if models are considered in isolation, they may require user input at run-time when the system changes or to avoid inconsistencies [FKN⁺92]. To facilitate coordinated and synchronised analysis across models, these should have semantic aspects in common and also closely integrate with the system's code.

1.1.3. R3 - Composability

Composable models are needed to enable automatic construction and analysis of models from representations of each component. If architectural reconfiguration is conducted using non-composable models, configurations involving components only discovered at run-time cannot be considered as the models associated with the new configuration cannot be created without additional user input. Composability is also necessary to reduce the specification burden for complex systems by enabling individual modelling of each component, thereby facilitating incremental elaboration and the definition of more fine grained representations.

1.1.4. R4 - Scalability

When selecting a new architectural configuration, the corresponding analysis needs to scale well in order for the benefits of choosing the configuration that maximises non-functional properties to overcome the cost of analysis and adaptation. In some application domains the environment or operational conditions may vary significantly, thereby requiring adaptation to be applied in few seconds. Other domains may not have enough resources to support analysis of non-functional properties using composable models. Therefore, composable representations may need to be minimised before composing in order to mitigate state-explosion associated with concurrent models. But reducing probabilistic models can be challenging. For example, when deleting a probabilistic transition what should happen to the remaining transitions?

1.1.5. R5 - Adaptation Safety

When a given valid architectural configuration, *i.e.* based on compatible bindings between provided and required interfaces, is selected for reconfiguration, it may lead the system to inconsistent states due to behaviour mismatch related to component's interactions. For example, functionally equivalent components can implement different versions of the same functionality, which may be based on different assumptions w.r.t. interaction with other components. In addition, the system configuration cannot be changed whilst the components to be replaced are being used by others. Consequently, reconfiguration mechanisms need to ensure that the new configuration does not contain behaviour mismatch and also that the system is led to a state where it can be adapted.

1.1.6. R6 - Decentralisation

Architectural reconfiguration may be required to be based on distributed or even on fully decentralised processes to avoid relying on a single physical host. A centralised solution may not be feasible for other reasons, *e.g.* commercial, to avoid that a single node possesses knowledge about all components of the system. Therefore, models of architectural, behavioural, non-functional properties need be compositional in order to support analysis without requiring the computation of the full model of a system configuration.

1.2. Contributions

The objective of this thesis is to define a framework that supports automated architectural re-configuration by integrating compositional models of architectural, behavioural, non-functional properties of component-based systems. Therewith, this thesis makes the following contributions:

Probabilistic Component Automata The primary contribution of this thesis is the definition of a composable modelling formalism, Probabilistic Component Automata, to represent the probabilistic behaviour of component-based systems. We define mechanisms to combine architectural, behavioural and non-functional aspects to support automated analysis of reliability requirements. Our model establishes a close link with Architectural Description Languages, such as Darwin [MDEK95], to describe the system behaviour. The representation for a composite component is automatically constructed by combining the architectural model of a given configuration and the PCA model of each component. PCA also provide suitable semantics to accurately represent the behaviour of software components by establishing a close link with architectural aspects and object-oriented languages.

Compositional Reliability Analysis Our second contribution consists in a novel algorithm to remove internal behaviour of a PCA model, thereby mitigating the state-explosion problem associated with composite models. This algorithm extends the concepts of Compositional Reachability Analysis to probabilistic models to define how internal actions are removed and, more importantly, how their probabilistic information is propagated to remaining transitions. The scalability of reliability analysis using composable models is greatly improved by reducing the PCA representation of each component before composition. We have further demonstrated both empirically, and formally using the notion of weak bisimulation, that the reduced models produced by the algorithm preserve the reachability properties of the original representation. As a result, the analysis of reliability and operational requirements that do not involve internal actions is equivalent for the original and reduced models. Properties that involve specific internal actions can still be analysed as the algorithm takes as input a generic set of internal actions to be removed.

Multi-view Framework The third contribution of this thesis is a multi-view modelling framework that leverages the integration that we establish for architectural, behavioural and management models. The framework includes two algorithms for architectural reconfigu-

ration driven by reliability requirements, which enable systems made of autonomous components to re-configure themselves to preserve global non-functional requirements. The required representations for selecting a new architectural configuration are first reduced according to the bindings between components in order to remove internal actions as well as behaviour associated with unused functionality. As a result, the selection process can involve different entities as the internal behaviour of each component is not disclosed, but only the choices that a component makes over its bindings.

Tool Support Finally, both the PCA formalism, associated operators and the aforementioned reduction algorithm have been implemented in an extended version of LTSA [MK06], a well-known modelling and analysis tool for LTS representations.

1.3. Thesis Outline

The remainder of this thesis is structured as follows. In Chapter 2 we review the different approaches that underpin adaptive software architectures. We first describe adaptation mechanisms to support changes in the behaviour and structure of components. We then discuss formalisms to model the structure and behaviour of component-based systems, the self-managed frameworks that use these models and the adaptation mechanisms to perform adaptation at run-time. While automating adaptation driven by non-functional properties requires combining structural, behaviour and non-functional aspects of a system, existing approaches are either based on textual annotations of non-functional properties or rely upon non-composable models which do not consider structural aspects.

To this end, in Chapter 3 we present Probabilistic Component Automata (PCA), the associated operators and their corresponding semantics to model the behaviour of basic and composite components, including explicit mechanisms for failure modelling, handling and propagation. The last part of the Chapter covers the semantics of the parallel composition operator and how it is combined with the architectural model to automatically construct the composite models.

As composite representations may suffer from state-explosion due to modelling of concurrent execution, in Chapter 4 we propose an algorithm as an extension to Compositional Reachability Analysis for reducing a PCA model by eliminating transitions associated with internal actions and propagating their probabilistic information to the remaining transitions, when possible. We illustrate using a set of simple examples the gains produced by the algorithm in computational

complexity for composite model construction and analysis of reliability requirements. The reductions in model sizes obtained for those models are in excess of 80% which leads to a reduction in excess of 90% in the time to analyse reliability properties.

While we have empirically verified that the reduction algorithm preserves the reliability properties of the original models of the examples in Chapter 4, in Appendix A we review standard notions of behaviour equivalences and extend them to formally show that the minimised model produced by the reduction algorithmic preserves the properties of the original model. Specifically, we show that the reduced model is weakly bisimilar to the original one.

In Chapter 5 we describe architectural reconfiguration processes underpinned by the architectural, behavioural and probabilistic models discussed in the previous chapters. Building upon a multi-view framework that integrates these models, we first discuss a centralised version of the reconfiguration process based on a central point that possesses knowledge about the available components for reconfiguration and their associated models. The reduction algorithm presented in Chapter 4 along with the operators discussed in Chapter 3 are used to automatically construct and analyse the composite representation for each possible architectural configuration. We then propose a distributed version which enables each component to make local choices regarding its required functionality and/or internal configuration. The reduction algorithm is used to compute the information exchanged by components that underpins local choices, without disclosing information about internal configuration encapsulated in the architectural model.

Finally, we illustrate the work presented in this thesis using an e-banking system from Java EE examples. This example system provides a simple but realistic application, of a reasonable size and with several sources of failure. We discuss how the behaviour of each component/class implemented in Java is modelled using PCA and how probabilities of internal, output and failure actions can be automatically extracted from execution traces. We then construct the composite representation and show how it is used to analyse relevant operational and reliability properties.

In Chapter 7 we summarise of the work presented in this thesis, review the contributions in light of the defined requirements for self-managed component-based systems and discuss possible further work.

Finally, we present in Appendix B our extension to the LTSA which supports specification, construction and analysis of PCA models.

2. Background

After introducing in the previous Chapter the general motivation and main requirements for adaptive software architectures, we review in this Chapter the relevant literature on self-managed software systems. We start by reviewing programming language support and other low-level techniques to adapt the system behaviour and/or structure in Section 2.1. We then discuss in Section 2.2.1 architectural models that cater for modular representations of the system structure which enable architecture adaptation based on adding and replacing components. Although some models associate textual notations of non-functional properties with components, automating architectural reconfiguration requires reasoning over formal representations of non-functional properties to accurately distinguish between components with equivalent functionality. These are discussed in the next Chapter in more detail. In Section 2.2.2 we discuss behavioural formalisms that enable analysis of the soundness of each configuration based on representations for each component that model internal behaviour and interactions between components. We then present in Section 2.3 different methods to specify when and how the system should adapt as well as management frameworks that provide re-usable services, such as monitoring and execution, to support adaptation. In Section 2.4 we describe methods that determine when the system, or parts of the system, can be stopped and adaptation can be applied whilst ensuring the system correct functioning. We conclude this Chapter by evaluating the limitations of the presented approaches w.r.t. to the requirements discussed in the previous Chapter.

2.1. Adaptation Techniques

Systems need to adapt in order to cope with changes in the environment and operational conditions, but the extent to which the system adapts can vary. The simplest technique to adapt a system is by changing the value of its configuration parameters. For instance, a video application can accommodate different traffic conditions by changing the parameters of TCP, *e.g.* window size, or by adjusting the video quality. Other examples exist for adaptation of specific

applications. However, these techniques are usually too low-level and specific to a particular application or domain. In the next Sections we cover higher-level techniques to change the behaviour/structure of an application/system.

2.1.1. Programming Languages Support

Programming languages comprise means to control dynamic objects at runtime that can be used to support adaptation techniques. For instance, Erlang includes the *hot code upgrade* feature that allows to change a program code at runtime. Erlang code is loaded as module units and the runtime system supports the simultaneous existence of two versions of a module in memory. The versions are referred as *old* and *new* and running processes can concurrently execute both versions. A process switches from the *old* to the *new* version by making an external call, when it conducts possible state transformations to use the *new* version. This allows a system to adapt with no interference. Furthermore, an Erlang application is structured as a *supervision tree*, where processes are hierarchically organised and divided in two groups: *workers* that perform computations and *supervisors* that monitor and manage *workers*. Each *worker* is structured into two parts: a *behaviour* that represents the provided functionality, and a *callback* that consists in a specific implementation of such functionality. ContextErlang [GPS10] is a context-oriented extension to Erlang in which applications consist of a set of components, each one having a single behaviour module and various callback modules that can be dynamically activated and deactivated based on context changes. More than one callback module can be active at the same time in a given component, as the set of implemented functions are not necessarily identical. When active callback modules have overlapping functionality, the order of activation determines which callback module is used.

Dynamic class loading supported by modern Object Oriented languages, such as C# and Java, in conjunction with reflection mechanisms can also be used to enable late bindings of classes not known or available at design and compile time. At design time, dependency injection mechanisms allow the client part of an application to delegate to an external configuration module the choice over the specific library or object that an instantiation of the client program uses.

2.1.2. Interception as a Means to Implement Adaptation

Middleware layers allow to introduce mechanisms to transparently intercept messages/invocations between components in order to perform adaptation, filtering or forwarding to other components without the original ones being aware of such actions. In the next paragraphs we review some examples of these mechanisms for CORBA and Java platforms.

Adaptive CORBA Templates (ACT) [SM04] rely upon the concept of a generic interceptor, which is a particular type of a CORBA portable request interceptor that adds adaptation capabilities to an existing application without having to change and recompile its code. A generic interceptor is registered in the ORB during its instantiation, intercepts all incoming and outgoing requests and sends them to the ACT *core*. The ACT core comprises one or more rule-based interceptors that perform a given action when a condition is verified. Examples of possible actions are: send new requests; record statistics; raise a *ForwardRequest* exception in order to forward the request to another CORBA object. This object is a surrogate for a CORBA object that provides the same set of methods, using a different implementation.

Another form of realising dynamically activated and transparent adaptation is by extending virtual machines with means to intercept and redirect interactions with the application code. *Guaraná* [OB99] achieves this by extending the JVM code, whereas *Iguana/J* [RC02] employs aspect-oriented techniques to extend the JVM without modifying its source code. Although these approaches perform adaptation in a transparent way in respect to the application code, the required modifications of JVM reduce their portability.

TRAP/J [SMCS04] applies similar (aspect-oriented programming) techniques to transparently generate adaptable Java programs at compile time. By using Aspect/J, an aspect-oriented extension to Java, TRAP/J does not require the JVM and the application source code to be changed in order to dynamically adapt an application. Wrapper- and meta-level classes are produced by an Aspect Generator and a Reflective Class Generator for each of the classes that need to be adapted. These are then *weaved* together with the application source code by the AspectJ compiler to construct an adapt-ready application. New functionality can be introduced at runtime by using *delegate* classes which contain a set of the methods of the application base code. Meta-level objects dynamically redirect messages originally sent to a wrapper-class to the corresponding delegate class.

2.1.3. Proxies to Implement Adaptation

Explicit proxy components can be placed to encapsulate functionality and intercept the interactions from other components to perform behaviour adaptation. This technique requires the application source code to be adapted to support indirect interactions through proxy components.

Cavallaro *et al.* proposed using proxy services to find a substitute a web-service when the previously used becomes unavailable. For each functionality required by the system, a proxy service receives clients requests to the faulty service and forwards them to an alternative service that performs the same operation. Similarly, Contract-based Adaptive Software Architecture (CASA) [MG05] uses *Handle* classes, which implement the same interface as the specific component class, to support adaptive selection of different implementations of the same functionality. The operations invoked on the *Handle* class are forwarded to the currently used implementation through structural reflection. When a class instance is replaced, its internal state is exported using a common representation and transferred to the new instance. In addition, each class needs to define safe points where its execution can be suspended when it needs to be replaced, and where the new instance can be started. Where the definition of such points is infeasible, the *Handle* class waits for the conclusion of the current invocations and buffers the incoming ones until the replacement takes place.

2.2. Software Models

While the previous mechanisms provide the means to enact adaptation at lower levels, they do not cater for the representation or analysis of the architectural and behavioural aspects of a system which are required for automating adaptation processes.

2.2.1. Architectural models

Architecture Description Languages (ADLs) are declarative languages to describe the structure of a system based on functionality provided and required by components and their connections. Existing approaches can be divided in two groups: those that consider connectors, which comprise a specification of behaviour interaction between components, as first-class entities equivalent to components, and those that leave such specification to the component's implementation. Architecture models can include descriptions of architecture styles for a group of systems that denote

reusable architectural specifications and determine the allowable structural modifications that can be conducted at runtime. Component-based architectural descriptions are composable as the structure of complex systems can be defined based on bindings between basic and composite components, which can themselves be composed of basic and composite sub-components and respective bindings between them.

UniCon [SDK⁺95] and Wright [AG94] are examples of ADLs that require interaction between components to be specified using connectors. A connector specification consists in roles that define the behaviour of participant components in an interaction. The interaction is mediated by glue code that constrains the interactions between roles. A component is described as a set of ports and a specification of its functionality. C2 [Med96], another ADL example, was originally defined as an architectural style for Graphical systems. It then evolved to an ADL where components and connectors have only a top and a bottom ports, resulting in strict hierarchical architecture designs. Communication between components is only achieved via connectors as components are not allowed to interact directly, while connectors can be connected to each other. Both components and connectors are permitted to be bound as follows: the top of a component can be attached to the bottom of a single connector; the top of a single connector can be attached to the bottom of a component; two connectors can be attached to each other from the top of one to the bottom of the other. There is no limit on the number of connectors and components that are attached to a single connector. Moreover, a component is only aware of other components that are *above* it in the architecture hierarchy, having no knowledge of components *beneath* it. An *interface definition language* is used to express the public functionality and data of a component, based on the following elements: *a)* top and bottom notification handling and request sender interfaces; *b)* initialisation parameters; *c)* publicly available methods; *d)* component semantics using behaviour specification; *e)* description of component context. An example architecture of a video game described using C2 ADL is shown in Figure 2.1.

While the previous ADLs comprehend explicit specifications of components interactions through connectors, the following ADLs delegate such specification to the component's implementation. Rapide [LKA⁺95] focuses on providing constructs to express architectures in an executable form for behavioural analysis through simulation at design stage. An architecture in Rapide consists of components having modules and an interface to specify required and provided functions from other components modules. Connection rules determine the communication between interfaces and formal constraints dictate legal and illegal communication patterns. Architectural con-

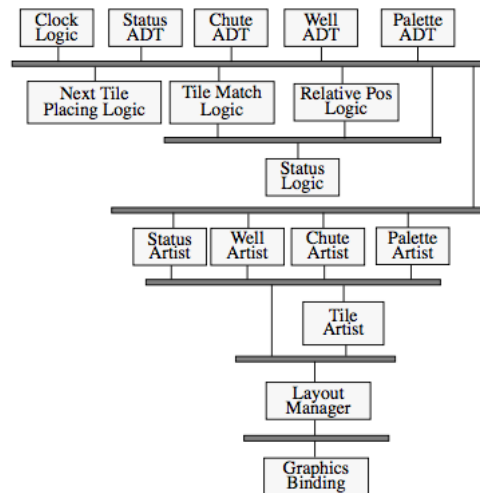


Figure 2.1.: Example Architecture of Video Game using C2 ADL [Med96]

straints are defined using conformance rules that state which components can be connected to a given component interface. Rapide separates the representation of the above constructs in five languages: 1) a *type language* to characterise components interfaces; 2) an *architecture language* to denote the communication between components; 3) a *specification language* to describe components behaviour; 4) an *executable language* to define executable modules in components; 5) and a *pattern language* to express allowed patterns of communication events.

Darwin [MDEK95] is another declarative language to specify the architecture of distributed component-based systems which does not distinguish components from connectors. Each component specifies the set of services it provides to others and the set of services it requires from other components, each one having a certain type. Components interactions are defined by one-to-many mappings between provided and required interfaces, *i.e.* a provided port may have many required ones bound to it whereas a required port can only be bound to one provided port. Furthermore, a composite component defines composition by establishing bindings among internal components ports as well as binding from its ports to the ones of internal components. Darwin also comprises the definition of dynamic architectures that change at runtime through dynamic instantiation. This construct enables the specification of components that can evolve arbitrarily or in response to a change in the environment. For instance, a composite server component can create new message handler sub-components in response to a request from the system manager. Since dynamically created sub-components instances are anonymous at design time, the type of the sub-component is used to specify internal bindings instead of the component instance name.

Acme [GMW00] is as a second-generation ADL which can also be used as an interchange language [GMW97], *i.e.* a common representation for software architectures specification that provides support for the use of various existing analysis tools of different ADLs, such as Wright and Rapide [GMW97]. The foundation concepts used for architectural description are based on the ones shared among previously defined ADLs. Components represent computational and data elements of a system having one or more interfaces, designated as ports. Each port defines a point of interaction and may be represented by a method signature. Connectors manage interactions between system elements and have interfaces designated as roles. A role establishes a participant of the interaction mediated by the connector. A system is defined as a graph in which nodes designate components and edges denote connectors and represent connections between pairs of a component's port and a connector's role. Acme introduces low-level internal descriptions of a component or a connector, denoted as representations. Each component or connector may have different representations to express its multiple views. Representation maps specify the correspondence between an internal representation and the external interface. For instance, defining the association between internal and external ports. Each of the previous architectural entities can be annotated with a list of properties, each one characterised by a name, an optional type and a value. However, the semantics of these properties is specific to external analysis tools and has no interpretation in Acme.

While the previous ADLs are independent from implementation languages which may lead to the violation of architectural structure and other inconsistencies, ArchJava [ACN02] is a Java extension to define a software architecture and seamlessly integrate it with implementation code in Java. The implementation has to comply with the *communication integrity* property, *i.e.* implementation code of components only interacts directly with components to which there are bindings in the architecture specification. The architecture is specified in terms of components with provided, required and broadcast ports as well as connections between ports. While *require* ports only support single bindings, broadcast ports may be connected to various provided ports, though these must return *void*. This is because when a broadcast is *invoked* it results in simultaneous calls to multiple provided interfaces. In addition, components can be dynamically created at runtime using the operator *new*, triggered by the invocation of a given method. These components can be bound at runtime using the *connect* operator. The *connection pattern* expression can be used to explicitly express the allowable bindings for dynamically created components.

Architectural models enable the representation of the system's structure as a composition of basic and composite components, instead of having a monolithic design. This modular structure supports easy replacement of independent sub-parts (components), which can also be reused in different systems. Although some models support reasoning on the interactions between components through connectors, the architectural models described in this Section do not cater for analysis of the system behaviour based on a composite representation constructed from a representation of each component.

2.2.2. Behaviour Models

Reconfiguring the system architecture at run-time may involve considering configurations which include components not known at design-time or from third-party providers. However, these components may lead the system to inconsistent states which can result in serious incidents, *e.g.* caused by a deadlock. In this Section we present behaviour models that support analysis of *safety* and *liveness* properties of each architectural configuration using the corresponding composite model. Such analysis can be used to detect inconsistencies before adapting the system configuration.

Labelled Transition Systems (LTS) [MK06] support modelling, construction and analysis of distributed component-based systems¹. Aspects related to data representation, resource allocation and user interaction are abstracted by LTS models in order to focus on concurrency details. Consider the execution of a concurrent program which consists of multiple sub-processes. When a (concurrent) program is executing, its state can be characterised by the values of explicit variables. The execution of a single statement transforms the program from one state to another. The sequential execution of each sub-process is modelled as a Finite State Machine in the form of LTS representations, whereby each transition between states is labelled with actions that represent atomic statements, though variables of the system do not need to be explicitly encoded. Formally, an LTS is defined by the following tuple $\langle \mathcal{S}, q, \mathcal{E}, \Delta \rangle$, where

- $\mathcal{S}$ is a finite set of states, q is the initial state,
- $\mathcal{E}$ is a set of action labels, and

¹Calculus of Communicating Systems (CCS) and Communicating Sequential Processes (CSP) are examples of formal languages that enable the specification of the behaviour of concurrent systems. Although both CSP and CCS explicitly distinguish between the receiving and sending ends for synchronisation between processes, the existing operational semantics [CS99] are equivalent to the ones specified for LTS models.

- $\Delta \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ is the set of transitions, each defined by a source state in $\mathcal{S}$, an action label in $\mathcal{E}$ and a destination state in $\mathcal{S}$.

As LTS are graphical representations, Finite State Processes (FSP) are used as a textual notation for describing behaviour. For every expression E in FSP, there is a corresponding LTS model A as defined by function $lts : E \rightarrow A$. The basic operators supported by FSP to specify the execution of sequential processes are prefix and choice. Analogously to other process algebras, FSP has algebraic properties as expressed by operational semantics of its operators.

The prefix operator defined in Rule 1 describes a process that initially executes action a and the subsequent behaviour is described by expression E .

$$\frac{}{(a \rightarrow E) \xrightarrow{a} E} \quad (\text{Rule 1})$$

The corresponding LTS is given by $lts(a \rightarrow E) = \langle \mathcal{S} \cup \{p\}, p, \mathcal{E} \cup \{(p, a, q)\} \rangle$.

$$\frac{}{(a_1 \rightarrow E_1 \mid \dots \mid a_n \rightarrow E_n) \xrightarrow{a_i} E_i} \quad (\text{Rule 2})$$

The *choice* operator enables the specification of different possible outcomes from a given state, as defined by Rule 2. The expression $(a_1 \rightarrow E_1 \mid \dots \mid a_n \rightarrow E_n)$ describes a process that initially engages in any of the actions a_i , and then behaves as described by the corresponding FSP expression E_i . The corresponding LTS model is formally defined as follows. Let $1 \leq i \leq n$ and $lts(E_i) = \langle \mathcal{S}_i, q_i, \mathcal{E}_i, \Delta_i \rangle$, then $lts((a_1 \rightarrow E_1 \mid \dots \mid a_n \rightarrow E_n) \xrightarrow{a_i} E_i) = \langle (\bigcup_i \mathcal{S}_i) \cup \{p\}, p, (\bigcup_i \mathcal{E}_i) \cup \{a_1, \dots, a_n\}, (\bigcup_i \Delta_i) \cup \{(p, a_i, q_i)\} \rangle$. Additional operators are supported by FSP to specify more complex behaviours of sequential processes, *e.g.* conditional behaviour [MK06]. On the other hand, the representation of the execution of a concurrent program is automatically constructed by composing the representations of its sub-processes. The same principle can be used to model a composition of components from their individual LTS representations. Given two LTS models A and B , their concurrent execution is denoted by the parallel composition between them: $A \parallel B$. Interaction between two processes A and B is modelled through synchronisation of their shared actions. Rule 3 determines that processes A and B only interact when they are both ready to execute the shared actions.

$$\frac{A \xrightarrow{a} A', B \xrightarrow{a} B'}{A \parallel B \xrightarrow{a} A' \parallel B'}, a \in \mathcal{E}_A \cup \mathcal{E}_B \quad (\text{Rule 3})$$

On the other hand, non-shared actions are executed concurrently and all their possible interleavings are included in the composite model, as defined by Rule 4.

$$\frac{A \xrightarrow{a} A'}{A \parallel B \xrightarrow{a} A' \parallel B}, a \notin \mathcal{E}_B \quad \frac{B \xrightarrow{b} B'}{A \parallel B \xrightarrow{b} A \parallel B'}, b \notin \mathcal{E}_A \quad (\text{Rule 4})$$

Other operators such as *re-labelling* enable the specification of scenarios with shared resources, while hiding replaces action labels with the silent action τ to denote internal actions and avoid synchronisation with other components when constructing composite representations. LTS models with internal actions can be minimised using Compositional Reachability Analysis [CK96] in order to mitigate state-explosion associated with interleaving of non-shared actions. We will discuss this in more detail in Chapter 4.

All the previous operators for behaviour modelling and construction of LTS models are supported by the tool Labelled Transition System Analyser (LTSA) [MK06]. Safety and liveness properties can be expressed using Fluent Linear Temporal Logic [MK06] and automatically analysed using the model-checking mechanisms supported by LTSA.

While LTS models do not distinguish between input, output and internal actions, Input/Output (I/O) Automata [LT89] in contrast do but introduce a different operational semantics. An automaton can determine when it executes output or internal actions, but it is not able to block the execution of any input action. As a result, *output* actions do not block as in CCS (or shared actions in LTS) as they can be synchronised with the corresponding input action at any time. Consider $enable(s) \subseteq \mathcal{E}$ as the set of actions that can be executed from state $s \in \mathcal{S}$. The semantics of I/O Automata requires processes to be *input-enabled*, *i.e.* for each state $s \in \mathcal{S}$ the process needs to be able to execute all the input actions $\mathcal{E}^{in}$: $\mathcal{E}^{in} \subseteq enable(s)$. Consequently, even when a sequential process is executing internal statements it cannot block interaction with other processes until both are ready to synchronise. This does not always represent a suitable semantic for a software system. Note that multi-threaded reactive programs, which resemble the semantics of I/O, can also be modelled using the previous formalisms by means of different processes, each one denoting a specific thread.

Interface Automata (IA) [dAH01] is a modelling formalism based on I/O automata, though it does not require input-enabledness. Apart from the distinction between input and output actions, IA models have similar semantics to LTS models. One important distinction lies in the synchronisation of input/output actions which is only allowed between two components, whereas

LTS models support synchronisation on shared actions amongst more than two components. Although this restriction may limit the ability to model publish-subscribe type of interactions using IA, it is suitable for representing synchronous interactions such as method invocations. Furthermore, the algorithm that implements the parallel composition operator for IA models computes first the product automata and then prunes invalid transitions. In contrast, the parallel composition algorithm for LTS models does not need to construct the full product automata, which can be significantly large. The composite model is constructed using an iterative method whereby its states and transitions are defined by the operational semantics rules previously defined in this Section. In other words, starting from the initial composite state which denotes the initial state of the models involved in the composition, the operational semantic rules determine the valid transitions from each composite state.

Given two components with bound interfaces and the associated IA models, the corresponding composite model can be used to verify if the components are compatible w.r.t to their interaction through input/output actions. On this account, IA include a notion of refinement that enables valid replacement of components. An interface automaton Q refines an interface automaton P , denoted by $P \succeq Q$, if all the input steps of Q can be simulated by P and all the output steps performed by P can be simulated by Q , while the internal steps of P and Q are independent. A component whose behaviour is represented by IA P can be replaced by another component whose behaviour is denoted by IA Q if Q refines P ($P \succeq Q$) and Q is compatible with the environment that previously interacted with P . While the first condition resembles sub-class polymorphism and implies that any component can be replaced by another as long as it provides the same functionality, and possibly more, the second condition is needed to prevent any incompatibilities arising from the environment using the additional functionality of the new component.

Although the previous formalisms support automated construction of models of composite components, along with automated analysis of *safety* and *liveness* properties, they do not support the representation of non-functional properties such as reliability and performance which require modelling the duration of actions and/or their frequency of execution. Discrete-Time Markov Chains (DTMCs) have been proposed by Cheung [Che80] to model the system execution profile and calculate its reliability based on the reliability of its components. However, this approach faces several limitations. Firstly, the model assumes that components execute sequentially and thus cannot represent concurrent execution. Secondly, the DTMC model of a composite component cannot be automatically constructed from the models of its sub-components.

Thirdly, this approach assumes that failures occur independently in components bound to each other and cannot represent failure dependencies and failure propagation across component bindings. In Chapter 3 we discuss in more detail these limitations and other models for reliability analysis.

2.3. Management

While software models enable modelling and analysis of the system structure and behaviour, management services and frameworks determine when and how the system adapts. In this Section we first review the main decision-making mechanisms which are responsible for choosing the most suitable adaptation action to be executed in face of an internal or external phenomenon: the violation of a given goal or poor performance. We then discuss existing frameworks that implement the MAPE-K loop [IBM].

2.3.1. Decision-Making

Policies have been successfully applied to express automated management on distributed systems as well as dynamic changes in the system behaviour at runtime [Slo]. They can be characterised by the main three types: Action, Goal and Utility-Function policies [KW04]. Action policies specify the action(s) to be taken on the occurrence of an event, based on the current system state, and if a condition verifies (Event-Condition-Action), while Goal and Utility-Function policies specify a desired system state without defining how to achieve it. Goal policies denote desirable states that are either satisfiable or not, while utility functions assign a scalar value to a system state, with the most suitable state being the one with the highest value. These policies are more flexible than Action policies since the responsibility for determining the actions to achieve the desired states is left to the managing system.

Although a combination of these policies allows to express different concerns, it may create conflicts between policies. Generally, Goal and Utility-Function policies are less liable to conflicts since they are both expressed in terms of desired states. However, a set of Goal policies may not be mutually satisfiable. Similarly, various Action policies may be simultaneously activated and perform operations that change a common part of the system in a conflicting way. Utility-Function policies may introduce problems when combined with Action policies by operating upon the variables which appear in the Action clause. Nonetheless, Utility-Functions do not have a propensity to conflicts since the desired state is easily determined by the one with the

highest value. The drawback of these policies is the additional burden of exactly defining numeric values for the system state space. A common mixing pattern of different policies is using Goal policies as a constraint for Utility-Function policies [KW04]. In addition, Utility-Functions can be used to determine the choice among simultaneously activated Action Policies and are the basis for self-tuning mechanisms.

The next Section presents languages to express Action policies. Thereafter procedures that use Utility-Function policies are described. Finally, methods that deal with Goal Policies are discussed in Section 2.3.4.

2.3.2. Languages for Action Policies

SAFRAN [DL06] is an extension to FRACTAL components, described later in Section 2.5, to define adaptive applications based on Aspect-Oriented Programming (AOP) concepts and techniques that focus on a clear separation between business code and adaptation logic code. In AOP systems an *aspect* is a module that can alter the behaviour of a *base program* (business code). Each aspect combines (*pointcut*, *advice*) pairs, where a pointcut designates a set of points of interest in the execution of a base program (*join-points*) and an advice represents a code segment which realisation is triggered when the execution of a base program reaches a joint-point of the pointcut set. Due to the event-based style of adaptation procedures, SAFRAN adopts the notion of point-cuts as sequences of runtime events, including internal and external ones. Based on AOP concepts, the following are the main elements of SAFRAN:

- the *base program* corresponds to a FRACTAL component;
- *point-cuts* are denoted by internal and external events;
- *advices* represent behavioural reconfigurations;
- *aspects* specify adaptation policies that link joint-points to advices that can be dynamically *weaved*, *i.e.* dynamically loaded.

The FScript language [DL06] is used as the advice language to define adaptation actions, including a special notation based on the XPath language to easily navigate through the configuration of FRACTAL components, encoded as FPath. It relies upon a directed graph representation of FRACTAL components, where nodes designate components, their interfaces and attributes, and arcs correspond to relations between components including a type annotation. Adaptation

policies in SAFRAN follow the ECA pattern: when an internal or external event occurs, if a boolean FPath expression matches, then an action specified by an FScript reconfiguration is executed. External events supported in SAFRAN include:

- **changed(expression)**: perceives any change of the value of the expression, which can designate an attribute or a resource;
- **realised(condition)**: a specific case of **changed** construction that only discerns boolean changes.
- **appears(path)** and **disappears(path)**: detects the appearance or disappearance of a resource or an attribute; the path expression can be a generic one, *i.e.* not referring to a specific attribute/resource but rather a type, by putting the '*' character at the end.

Alternatively, the Stitch language [CG12] is based on a slight modification to ECA policies based on two constructions: *tactics* and *strategies*. Tactics implement the Condition-Action part of ECA policies where the condition can be composed by *exists* and *forall* conditions, and introduce an additional construct to indicate the expected behaviour of adaptation actions. Strategies are defined as a tree of Condition-Tactics nodes. Firstly, each strategy comprises boolean conditions activation, which can also include *exists* and *forall* conditions. Secondly, each node defines a condition for the tactic to be applied as well as an estimation of the time it needs to adapt the system. A list of conditional branches define the next steps in the tree, which can be one of the following: terminate in the current node if the tactic is successful, raise a failure exception or initiate another navigation from a given start node. The last construction also defines the number of allowed tree navigation loops. In addition, a tactic is associated with an impact vector which consists in a cost-benefit specification on system properties. A utility-function and weight is associated with each system property, which is then used to select the most suitable tactic.

In order to prevent long strategies from taking over the adaptation system, hence causing the delay of other adaptation decisions, a preemption mechanism was added to the Stitch language [RCGS10]. Each strategy is associated with a time utility curve (TUC) that expresses the expected utility of concluding a strategy depending on the time in which it is conducted. When there is a need for adaptation, *i.e.* a new strategy is triggered, and another strategy is currently executing, the decision-making service checks whether is better to let the latter strategy finish or preempt it and allow the former to execute. If multiple strategies are triggered, the decision-

making service selects the scheduling order that maximises a Predicted System Utility based on the TUC of each strategy.

2.3.3. Using Utility-functions

Utility-functions are high-level specifications for self-optimisation of software systems. They express the level of preference for a given system state according to some metric(s). One application of these specifications is to determine the most suitable adaptation action when more than one rule is triggered, based on non-functional metrics annotations. The utility value of an adaptation strategy can be computed using a weighted sum of several utility functions, each one using a given metric. For example, in [SHMK10] each component is annotated with non-functional metrics that correspond to its resource usage pattern, *e.g.* `cpu=low`, `memory=high`. Utility-functions rely upon those annotations to select the most suitable component when the previously used one fails or for optimisation purposes. As these annotations only correspond to predictions of system designers/administrators, they should be dynamically updated using the values collected by the monitoring service to reflect their real resource usage pattern [SHMK10].

Utility-functions are also used to express high-level business terms and service-level requirements that are used to dynamically allocate resources in a Data Centre management system [TK04]. In this case, the management system is structured as a two-level architecture: a low-level of logically separated Application Environments and a Resource Arbiter at higher-level. An Application Manager encapsulates the management details of the resources assigned by the Resource Arbiter by only providing to it the predicted resource needs. Each Application Manager controls the assigned local resources of the Application Environment and periodically optimises their control parameters C using a service-level utility-function R , based on the measured service level S and demand D aggregated by the Data Aggregator from its locally managed application servers. A Modeler handles a model of system performance $S(C, R, D)$ based on the current control parameters, allocated resources and service demand. The Utility Calculator computes the optimal resource-level utility considering an estimate of the predicted demand in conjunction with the all possible resource allocations. This function is periodically passed to the Resource Arbiter by all running Application Managers. The Resource Arbiter uses those functions to periodically recompute the allocated resources to each Application Manager, however cost associated with switching resources are not considered.

2.3.4. Mechanisms Supporting Goal Policies

In filling the expressiveness gap of Action Policies, Utility-Functions provide the means to indicate preference in terms of metric values. Goal Policies are based on boolean predicates that establish the system correct functioning according to its requirements. Although they do not specify actions for adaptation, these policies give the means to assess if a system is violating its specification. Given one or several goals, planning algorithms are used for the generation of action plans to fix the system relying upon a domain description as input, containing the available actions and information about the evolving environment.

Sykes *et al.* [SHMK07] generate reactive plans using the Model-Based Planner tool (MBP) [BCP⁺01], from a domain description defined in SMV, to perform architectural changes when high-level goals policies specified by the system designer are violated. The domain description includes state-predicates, pre- and post-condition constraints on the available actions, the initial state and a goal. The generated plan consists in a set of condition-action rules that comply with the following requirements: a condition refers to a state in the environment from which the goal is achievable and it is viable to apply the associated action in that state. Moreover, the set of condition-action rules can be represented as a graph of state transitions. During the execution of the plan, the shortest path to the goal is selected from the available ones to reach it until the goal state is achieved. Furthermore, the possible large size of state spaces in the definition of the domain description may hinder the applicability of planning algorithms. To address this issue, the domain model is structured as a hierarchy of partial descriptions and sub-plans are generated for each one, using a bottom-up approach. Finally, the required changes in the component configuration are automatically derived from the actions included in the plan. For instance, if the plan contains an action performed by a non-active component, that component is instantiated before plan execution.

2.3.5. Self-Managing Frameworks

Self-managing frameworks provide adaptation capabilities to software systems. The main functions associated with self-management are characterised by MAPE-K loop: Monitoring, Analysis, Planning and Execution, all underpinned by system Knowledge. Monitoring concerns with the supervision of the system and its surrounding environment. Analysis deals with the reasoning over system state and the environment conditions to detect functional inconsistencies as well as nonfulfillment of system goals and desired performance. On top of that, Planning builds

adaptation plans based on the reasoning conducted by Analysis. The generated adaptation plans vary from design-time plans for predicted scenarios and runtime plans for unpredicted situations. Finally, Execution is responsible for conducting the realisation of the generated adaptation plans and guaranteeing system consistency throughout plan execution. In the next paragraphs we discuss how existing frameworks realise the MAPE-K loop using the concepts and techniques presented in the previous Sections.

StartMX [AST09] is a centralised framework to create applications with self-managing properties for Java-based systems relying upon Java Management Extensions (JMX), as well as on a policy engine. JMX technology is used to provide the generic means to cope with different resources, to provide managing and monitoring interfaces for adaptation managers as well as a notification mechanism. A centralised execution engine supports different external policy engines or adaptation code written in Java. Policies can be dynamically loaded, unloaded and redefined. An adaptation procedure is represented by an execution-chain sequence of processes, each one representing a function of an autonomic manager (Monitor, Analysis, Plan or Execute). A process can interact with the managed resources using the managing interfaces provided by JMX.

Prism-MW is a framework supports dynamic software updates, decentralised service discovery, transparent replication and logical mobility [ESP⁺07]. An active deployment and analysis environment (*DeSi*) centralised point complements the Prism-MW with visualisation and management facilities for the software architecture. Each host that instantiates the Prism-MW has an *Admin* component that consists in a local management unit which is responsible for instantiation, addition, upgrade and removal of components residing on the host, by interacting with *DeSi*. Additionally, a Service Discovery Engine (SDEngine) maintains a database with all the services provided by components executing in a given host, together with an SDConnector to route service discovery requests to the host where a proper service provider resides. On top of that, a fault-tolerant connector is placed before the service provider and transparently applies a synchronisation mechanism among service replicas, *i.e.* instances of the same service type. However, the SDEngine of the service client is aware of the existence of the service replicas. If it detects that the main service provider has failed, it upgrades one of the backup replicas to the primary provider and informs all other SDEngines on the upgrade. Furthermore, the monitoring services integrated in each host report the values of non-functional properties concerning service execution to *DeSi*, which in turn applies optimisation algorithms to improve the

system architecture. The generated modifications are transmitted to the *Admin* components which coordinate the application of the corresponding adaptations. If these adaptations involve the replacement of a component for a new version, the *Admin* component transfers the state of the old version to the new one.

Rainbow is a centralised framework [GCH⁺04] that uses a generic dynamic architectural model to represent and manage a given software system. The architectural model is based on the Acme ADL [GMW00] annotated with non-functional properties, as well as architectural constraints. Monitoring facilities are used to dynamically update non-functional properties and to verify if the system is satisfying the specified architectural constraints. The framework is composed of various reusable units in each of its layers:

- *system-layer*: monitoring probes (sensors), resource discovery service and effectors, *i.e.* interfaces to adapt the system;
- *architecture-layer*: gauges aggregate info from probes and update the architectural model, which is handled by the model manager; a constraint-evaluator analyses the architectural model consistency and the adaptation engine verifies the need to adapt and decides the most suitable actions, which are directly conducted on the managed software system;
- *translation infrastructure*: bridges the abstraction gap between the system and the architectural model by mapping reusable units to the concrete system structure;
- *system-specific adaptation knowledge*: system operation model that defines parameters such as component types and properties, behavioural constraints and adaptation strategies specified using the Stitch language [CG12], to add self-management capabilities to the managed system using the adaptation infrastructure.

The system layer is responsible for keeping the runtime architectural model updated with the managed system throughout its execution. Although the constraint-evaluator examines the runtime architectural model to trigger Stitch strategies, once a strategy is activated it is directly executed in the software system.

The GRAF centralised framework [DAO⁺11] also relies upon a runtime abstract model between the managed software system and the centralised adaptation manager. However, adaptation actions in the GRAF framework are not directly applied to the managed software system in order to reduce the risks of adaptation. To this end, the framework architecture is organised into three layers:

- the Adaptation Management Layer comprises a Rule Engine which evaluates the adaptation policies in a repository and a Control Panel to add, edit and remove policies to/from the repository;
- the Runtime Model Layer maintains a repository of model history and controls a Model Manager that handles the Runtime Model representation and evaluates model invariants;
- the Adaptation Middleware Layer bridges the gap between the Adaptable Software and the Runtime Model layer using state variables adapters and model interpreters.

The control loop in GRAF is realised as follows. The state of the running adaptable software is disseminated to its runtime model, with such changes being realised by the adaptation manager through sensing the runtime system. By analysing that information, the adaptation manager chooses the most suitable adaptation policy and sends it to the Runtime Model Manager. The Runtime Model Manager evaluates the pre-conditions of the adaptation policy before applying adaptation actions on the runtime model. Thereafter, the effects of the conducted modifications are validated using the post-conditions specified by the adaptation policy as well as the model invariants. If they do not conform with such constraints the Runtime Model Manager rolls back the performed alterations. Moreover, the state variables are updated using State Variable Adapters that are connected with the code of adaptable software. The modifications performed on the Runtime Model are then effectuated on the Adaptable Software at predefined points in the software control flow, designated as interpretation points, similar to pointcuts in AOP.

The previous frameworks rely upon a centralised representation of the system and a centralised adaptation manager which may not scale for complex enterprise systems. iManage [KCES07] is a policy-driven modelling framework that partitions a system state space into smaller units to facilitate system management by reducing the complexity of building the system models and specifying adaptation policies for each small unit. Firstly, the framework provides tools to collect system parameters and metrics that are mapped into a single representation of the system state-space, which is then partitioned into small state units. Each unit includes *homogeneous variables*, *i.e.* variables which states are related to each other, while minimising the number of *knobs*, *i.e.* variables that the external change of their values affects the state of other values, which in turn affect the state of other variables. Another criterion used by the partitioning algorithm is guaranteeing that *knobs* required by one partition are not needed by others, in

order to have each partition orthogonal to the others. A partition establishes a *micro-model* with particular monitoring specifications, adaptation policies and goal rules.

The system state-space model and the micro-model are then used to derive ECA policies from a Goal policy. The framework registers a trigger for capturing a state where the goal is not being met, which is registered as the *Event* part. When the trigger is activated, the current small-unit state is recorded as the *Condition* part. A corrective action is then derived using the current small-unit state and it is stored as the *Action* part [KCES07]. Furthermore, as the state of micro-models evolve overtime the expected outcome of adaptation policies may not be consistently the same. On that account, each policy is associated with a confidence attribute that reflects the probability of the policy reaching its expected outcome. After executing a policy the framework evaluates its outcome and updates the confidence attribute accordingly. A threshold is specified to establish the minimum confidence level to allow the execution of a policy.

The Self-Managed Cell (SMC) [SL10] is a distributed self-management framework which encapsulates an administrative domain composed of a set of components. It comprises an extensible set of services which is managed by the *service discovery*. This service finds neighbouring components that can be added to the SMC. The management of the administrative domain is based on a policy engine using authorisation and obligation policies, *i.e.* ECA policies. A publish-subscribe event bus is used as the form of communication between services as well as SMCs. Obligation policies are triggered by events diffused by the bus.

2.4. Adaptation Correctness

Adaptation processes must not lead the system to an inconsistent state, an unstable situation or one where its specification is not met, *i.e.* the system correct functioning must be preserved. The formalisation of the adaptation process allows system designers to prove the correctness of the adaptation actions at design time.

An adaptation process can be formalised as a set of safe adaptation sequences of safe transitions between safe configuration states [ZCYM]. Those states are generated from the specification of system invariants and dependency among components. Each safe transition can be composed of several adaptation actions that are divided in three parts: a *pre-action* preparation phase, an *in-action* phase that changes the system structure and final tasks in the *post-action* phase. When the system detects a condition for triggering adaptation, a centralised adaptation

manager obtains the target configuration and prepares for the adaptation in the following three stages:

- using the source-target configuration and dependency relationships among components, a set of safe configurations is constructed;
- from the constructed set of safe configurations a safe adaptation graph is produced, having safe configurations as nodes and adaptation steps as edges;
- the solution with the minimum cost is found executing the Dijkstra's shortest path algorithm on the safe adaptation graph.

Alternatively, the concept of *quiescence* is used to define conditions to ensure that the adaptation process is properly conducted in a distributed system without violating its consistency [KM90]. A system is modelled as a direct graph in which nodes are system entities where communication is based on transactions and edges denote connections between these entities. The state of an application is abstracted into a set of different configuration states, considering *active* and *passive* ones as the two main states of each node. If a node is in the active state it can initiate, accept and service transactions. Although in the passive state a node still has to accept and service transactions, it cannot be involved in a transaction of which it is the initiator and cannot also start new transactions. Notwithstanding that the passive state is considered a necessary condition to ensure adaptation correctness, it is not sufficient since a node may still be involved in transactions of which it is not the initiator. Accordingly, the *quiescence* state is introduced as a stronger condition state for adaptation as follows: the node is in passive state, it is not involved in other transactions and no transactions that involve this node have been or will be initiated.

Furthermore, the quiescent state is proven to be achieved in bounded time [KM90]. Firstly, a node N has to move from active to passive state in order to reach the quiescent state. It reaches passive state in bounded time since it can finish in a certain limited interval the transactions it has initiated. Other transactions also terminate in a tied period since a node in passive state can accept and service transactions, provided that they do not lead it to start new ones. Secondly, in order to reach quiescent state, not only the node N must be in passive state but also the set of nodes T_N that are directly involved in transactions with node N have to move to passive state as well. This is to ensure that all transactions involving node N can be finished while new

ones are not initiated. As all nodes in T_N and N can reach a passive state in bounded time, node N is able to reach quiescent state in bounded time.

However, Vandewoude *et al.* [VEBD07] argue that the *quiescence* criterion is highly disruptive to the system execution. They have suggested using a low disruptive criterion underpinned by the concept of *tranquility* as an alternative to *quiescent* [VEBD07]. This criterion for adaptation correctness relies upon the following two considerations:

- If a node has finished its participation in a transaction, it can be adapted before the transaction completes. Similarly, if a node is involved in a transaction in which it has not yet participated, it can be adapted.
- As opposed to the underlying model of quiescence where the initiator of a transaction is aware of its completions, this model nodes follow a black-box design, hence the initiator of a transaction is only aware of its directly connected nodes. Although each participant in the transaction can initiate new ones in response to messages concerned with the transaction, the initiator of the original transaction is not aware of these *sub-transactions*.

The concept of tranquillity can be summarised as follows: a node is in tranquil state if 1) it is not involved in a transaction that it has initiated, 2) it will not start new transactions, 3) it is not handling a request and 4) its adjacent nodes are not involved in any transaction in which the node has participated and may still participate in the future. The tranquillity criterion for adaptation correctness is less disruptive than quiescence as only the node where adaptation is taking place has to be in passive state. Although the third condition for a node to be in tranquil state requires that some adjacent nodes finish transactions with that node before it can be adapted, those adjacent nodes do not need to be fully *passivated*.

The previous conditions establish when adaptation can be correctly applied without compromising system consistency. There is a trade-off between the level of assurance at design time of adaptation processes, which may lead to poor performance, and following an optimistic approach whereby adaptation is performed when needed and additional techniques are used to detect and fix instability situations related with the adaptation process. An important benefit of the approaches presented in this Section is that they are independent from the adaptation process.

2.5. Integrated Models

Separation of concerns is a principle widely applied in the development of complex systems, whereby the system functionality is decomposed into smaller units or different concerns in order to reduce the implementation complexity and to allow focusing on a particular concern. The same principle is applied to system models to enable models that focus on a particular aspect of a system. On this account, the Viewpoints framework [FKN⁺92] supports the specification of multiple-views to express different perspectives of a system. Although each view can be seen as a *vehicle for separation of concerns*, the different views need to be integrated to avoid inconsistencies when analysing the system. To this end, the Viewpoints framework supports the following relationships between viewpoints:

- independent views which are required by a project;
- non-overlapping views where there is some dependency on each other;
- partially or fully overlapping views.

The framework also includes mechanisms for interaction and cooperation between views that have dependencies or overlap in order to transfer information between different views and to check their consistency.

The same principle of *divide to conquer and unite to rule* [FKN⁺92] is also required for self-management system where the architectural, behavioural, non-functional properties and management concerns need to be combined to support automated adaptation mechanisms. In this Section we discuss approaches that combine models and techniques presented in the previous Section for integrated analysis of the system structured, behaviour and management aspects.

2.5.1. Architecture and Behaviour models

Most existing Architectural Description Languages described in Section 2.2 include formalisations of architectural styles or of interactions between components. The C2 Architectural style rules are defined using *Z* notation as a set of logical rules regarding architectural elements configuration and connections [Med96]. Additionally, Acme comprises a constraint language based on first order predicate logic to express design constraints regarding an architectural specification. In addition to the standard set of constructs related to first order predicate logics, the constraint language incorporates a set of boolean functions concerning specific aspects of architecture configurations, which include special predicates to verify if two components are directly

connected through a connector or indirectly connected via intermediate connections, if an entity comprises a given property or a given type, the set of connectors of a given system, the set of ports and roles of a given component or connector, respectively, and all other aforementioned elements. Two types of constraints are distinguished: invariant as mandatory constraints that need to be verified in order to consider a given architectural specification valid and heuristic constraints for which only a warning is produced when they are invalid. Architectural styles are then defined based on structural types, *i.e.* types of components, connectors, ports and roles. Each one encompasses a type name, a list of its substructure, constraints and properties. A style is characterised by a set of properties, a set of constraints, structural types and a default structure.

Darwin [MDEK95] relies on π -calculus to formalise the behaviour of provided and required ports as well as bindings. These formal definitions are used as input for static checking of implementation conformance with respect to the specification. Additionally, Modes [HKMU06] extend the Darwin component model with a representation of the supported behaviours of a component, which are characterised by the correspondent interaction between required and provided ports. The interaction process between different behaviours is represented by Finite State Processes that define a set of scenarios in which the component can operate, *i.e.* a composition of all possible interaction sequences. Safety and liveness properties can be analysed using the corresponding LTS representations. Constraints on architectural styles and component's interactions are specified using the Alloy language [Jac02], in which each specification consists of signatures (component and field declarations), facts and predicates (constraints) and assertions (properties).

Similarly, Wright ADL supports Communicating Sequential Processes (CSP) to model the behaviour of components and connectors to analyse the compatibility between a component port and a connector role as well as to verify if the system configuration is free from deadlock. Additional specifications on non-functional properties are supported by Wright ADL [VOKK] to associate non-functional requirements to required interfaces and non-functional assurances to provided interfaces. However, these specifications are based on textual notations and are not linked to the behaviour of each component. Wang *et. al* [WPC06] also explored mappings between architectural styles and DTMC representations of the entire system, though these mappings must be manually defined and the resulting models do not include the internal behaviour of each component.

2.5.2. Architecture and Adaptation

Some architectural models also provide support for architectural adaptation mechanisms. The distinction between provided and required interfaces plays an important role in enabling a clear separation between the component's functionality and management/adaptation concerns as it allows a third-party (external) component to determine the bindings between components.

In addition to encoding component's structure using provided and required interfaces, FRACTAL components [BCL⁺06] also include in a *membrane* that provides external control interfaces to introspect and reconfigure its internal details, and a *content* that consists in a set of sub-components. The membrane's control interfaces normally correspond to several controller and interceptor objects. These objects provide support for: *a)* obtaining the available external interfaces of a component; *b)* binding and unbinding interfaces; *c)* adding, removing or replacing sub-components; *d)* controlling behaviour, performing dynamic reconfigurations; *e)* starting or stopping execution. In addition, a sub-component can be shared among various composite relationships in order to preserve encapsulation, without having to deploy an external component to synchronise the sub-component state among all composite components.

Three levels of control are supported by FRACTAL components. The base level corresponds to components as black-boxes since no interception or introspection capabilities are provided. The next level corresponds to an interface that provides the means to find all external interfaces of a component (client and server), though no form of control is supported. The upper levels openly expose internal details, increased introspection and interception capabilities. The following are examples of controllers provided by FRACTAL that can be extended and combined to supply components with different reflective properties. Firstly, an *Attribute Controller* provides getter and setters for components attributes. Secondly, a *Binding Controller* allows to bind and unbind client interfaces to server interfaces. Thirdly, through a *Content Controller* one can list, add or remove sub-components. Finally, a *Life-Cycle Controller* gives means for controlling behaviour, performing dynamic reconfigurations or start or stop execution.

Furthermore, Sykes *et al.* [SHMK10] proposed a distributed algorithm to select architectural configurations based on non-functional properties associated with each component. An architectural configuration is selected based on an aggregated value of the non-functional parameters of its components. For each non-functional property, a monitoring service supplies the corresponding value provided by each component. However, the model does not consider dependencies between the non-functional values provided by a component and the ones given by the compo-

nents it requires. Grassi *et al.* [GMM13] extended Sykes' approach by defining non-functional properties based on the bindings between components. For example, the reliability of a component is given by the product between the reliability of its internal behaviour and the reliability provided by the components bound to its required interfaces. However, the reliability metric used assumes that all interfaces of a component are bound, and although not associated with a behavioural model of the components, does take into account the number of times components are invoked.

2.5.3. Behaviour and Adaptation

The MOCAS (Model of Components for Adaptive Systems) model [BHuB09] focuses only on behavioural adaptation relying upon the Unified Modelling Language (UML). MOCAS components communicate asynchronously by sending and receiving signals. Each component sets a UML state machine at runtime to characterise and realise its behaviour. This state machine consists in a set of states which are connected through transitions, each one being designated by an input signal, a guard (a boolean expression) and effects. A transition is triggered when the component receives its input signal and its guard is verified. Effects consist in internal actions to be performed as well as signals to be sent after the transition is handled. A state comprises *entry*, *do* and *exit* sets of actions that are conducted when the state is reached, when it is active and when it is exited, respectively. A state also includes invariants defined using OCL expressions which together with guards designate business properties.

In order to become adaptive a MOCAS component is wrapped by a MOCAS container that settles interaction between the component and its environment while providing adaptation capabilities [BHuB09]. Each component can have several sensors to monitor its environment. As each sensor is an adaptive MOCAS component, it can be started and stopped or even change its behaviour to be proactive, *i.e.* continuously monitoring the environment, or reactive, *i.e.* start monitoring the environment after receiving a certain signal. Sensors send their view of the environment through signals to an aggregator that centralises and reports sensed information to other components. These can be an adaptive MOCAS component or other aggregators from other components interested in that information. Furthermore, an evaluator component is responsible for deciding when to adapt the MOCAS component behaviour by using adaptation policies. Each adaptation policy is encoded using state machines similar to the ones described to characterise a component's behaviour.

Adaptive MOCAS components further comprise a coordination mechanism through connections among their aggregators to perform the following two types of coordinated adaptation:

- *reactive*: the component directly informs other components about its adaptation, which in turn can set off their own self-adaptation process though they cannot stop the one started by the first component. Components have to be hierarchically organised in order to avoid adaptation loops.
- *negotiated*: when an adaptive component needs to adapt, the most suitable adaptation policy is negotiated with other cooperating adaptive components.

Alternatively, feature-based approaches structure the functionality of a system as a set of features and alternative configurations are defined as different combinations of active features [EEM10, MLD⁺11a]. FUSION is a centralised framework [EEM10] that contains a learning module to compute the impact that each feature has on non-functional properties. The learning module relies on collected values of system metrics when each configuration of features is active. The adaptation manager is activated when an application goal is violated and a new feature configuration is selected such that it optimises the aggregated utility based on the learned functions for the impact each configuration on non-functional properties. When the goal can be achieved by more than one configuration of features, the impact functions are used to select the most suitable one. After adaptation, the difference between the expected and actual impact is measured to verify the accuracy of the learning process. When the difference is higher than a given threshold the learning process is repeated and the aforesaid functions are fine-tuned. Additionally, the enablement and disablement of features are the supported adaptation actions. The framework continuously monitors the running system and computes the corresponding utility. If a goal is violated, a new system configuration is computed in terms of enabled features that maximises the utility.

Although the feature-based adaptation system proposed by Modi *et al.* [MLD⁺11a] requires a mapping between a behaviour configuration and the corresponding values of non-functional properties to be manually specified, the selection of a new configuration considers deployment constraints as well as the costs associated with re-configuration. Deployment constraints define resource requirements for each behaviour configuration as a set of active features. For instance, at least 1MB of memory needs to be available, denoted as `mem(1MB)`. Each configuration is associated with corresponding qualitative non-functional values, which are mapped into numerical

values between 0 and 1 through fitness functions. Moreover, a model of application context is also considered using a DTMC representation, where each state represents a given context and transitions between states denotes the probability of the application moving from one context to another. Each context is also associated with utility values for non-functional properties in order to express user preference. For instance, in an emergency scenario response time may be more important than the screen resolution.

The adaptation manager performs the following steps when the application changes context or the current configuration is no longer feasible due to its deployment constraints. Firstly, a user benefit value, denoted by B_{curr} , is calculated for each feasible configuration based on the associated non-functional values and preferences of the current context. A future user benefit B_F is then calculated given the probabilities of moving from the current context to other contexts and the benefit of the behaviour configuration for each possible subsequent context. The current and future benefit are combined using an horizon h which controls the importance the application user gives to current context compared to future contexts: $B_{agg} = h \cdot B_{curr} + (1 - h) \cdot B_F$. Finally, the adaptation manager selects the behaviour configuration that maximises a trade-off defined by a parameter $alpha$ between aggregated benefit B_{agg} and the cost to adapt the system from the current configuration : $\alpha \cdot B_{agg} + (1 - \alpha) \cdot Cost$. Adaptation cost is defined based on the cost to deploy the new features in the candidate configuration and undeploy the features in the current configuration.

All the aforementioned approaches rely on non-functional parameters that are manually associated by a system designer as annotations to a model and are assumed to be correct. Filieri *et al.* [FGT12] have used DTMC models to automatically verify the reliability properties of a given system configuration. Profiling tools are used to keep the DTMC model of the system updated at runtime [EGMT09], thus enabling continuous verification of the reliability properties. However, as DTMC-based approaches treat components as black-boxes as the DTMC model of a composite component cannot be automatically constructed from the models of its sub-components. Consequently, such approaches can only be used in an re-configuration processes that relies upon alternative configurations defined at design-time. This limitation stems from the fact that a new representation has to be manually defined for each system architectural configuration. In the next Chapter we discuss in more detail different approaches for modelling probabilistic behaviour and reliability.

2.6. Background Summary

The adaptation techniques presented in Section 2.1 can be encoded at the architectural level through addition or replacement of components and changes in bindings between them. A *component*, in a generic sense, denotes encapsulation of behaviour, hence it can equally well be used for the representation of software libraries, hardware devices or (web) services. Therefore, architectural adaptation provides the right level of abstraction to enable the definition of adaptation mechanisms that are not specific to a particular programming language/execution environment. Interception and proxy-based techniques can be seen as lower-level mechanisms that realise architectural adaptation for a specific programming language or middleware framework.

In addition to being able to adapt their structure, autonomous systems need to determine when and how the system should adapt. Action policies consist in reactive rules which specify the exact steps for adaptation based on the current system state. However, such procedural constructs require specifying all the situations in which adaptation is needed. Declarative specification such as goal policies and utility functions are more suitable for autonomous systems as they enable dealing with unanticipated states that require adaptation. Action policies can be used to realise the plans generated to fulfil Goal or Utility policies.

Automating architectural reconfiguration requires integrating models of architecture, behaviour and non-functional properties to determine the possible alternative configurations, calculating their non-functional properties and choosing the most suitable configuration. Existing research does not fully address these aspects which are often left for subsequent design iterations or considered in isolation even though they are clearly interdependent. For instance, existing approaches that build upon architecture or behaviour models to generate alternative configurations that maximise aggregated non-functional properties rely upon textual notations of non-functional properties [SHMK10, MLD⁺11b] or monitoring components external to the system [CG12], which do not provide the desired level of accuracy for adaptation mechanisms. On the other hand, model-based analysis of reliability and performance using DTMC and CTMC representations do not support automatic construction of the composite model of each configuration from generic representations of each component. Consequently, only configurations known at design time can be considered for reconfiguration. Although composable models of system behaviour support verification of the *soundness* of new architectural configurations, the construction of the corresponding composite representations does not take into account the ar-

chitectural model, which can lead to incorrect analysis as the composite model does not reflect *active* behaviour.

In this thesis we aim to provide automated mechanisms for architectural adaptation based on a close correspondence between the architectural, behavioural and non-functional aspects of component-based systems. Composability is a key aspect that we aim to preserve across all modelling aspects as it enables automated analysis of architectural configurations not considered at design-time. We aim to describe architectural aspects using Darwin [MDEK95] a compositional for specifying the architectural structure of component based systems. The Darwin ADL also supports explicit specification of provided and required functionality of a component, thereby enabling an external (third-party) component to determine the bindings between provided and required of different components. We aim to automate architectural reconfiguration by delegating to a management component the decision to bound a provided interface of one component to a required interface of another. Additionally, we aim to describe the behaviour of each component using Interface Automata [dAH01], a composable formalism. Although IA and Darwin specifications have some modelling aspects in common, these formalisms are considered in isolation in existing approaches. As self-management requires a close integration between the architectural and behavioural aspects of a system, we aim to associate input-output actions with provided-required interfaces, and their synchronisation with bindings between provided and required interfaces. We aim to explore these links to support the construction of composite representations of the system behaviour that reflect the bindings between components. Moreover, we aim to use the composite representation of each system configuration to determine the compatibility between components using *safety* analysis tools in LTSA [MK06]. Although existing work includes composable formalisms for architectural and behavioural aspects, there is a lack of support for composable probabilistic models that support automated reliability analysis. To this end, we aim to fill this gap by proposing Probabilistic Component Automata (PCA), a compositional modelling formalism. We aim to extend Interface Automata with probabilistic information as well as support for the representation of failure scenarios, failure propagation and failure handling. By extending IA we aim to establish a close link between the behaviour representation and probabilistic models of a system and also leverage the aforementioned link between IA and Darwin models. By integrating architectural, behavioural and reliability composable models we aim to define centralised and distributed reconfiguration mechanisms that

enable systems made of autonomous components to re-configure themselves to preserve global non-functional requirements, while ensuring the soundness of new configurations.

We start by reviewing in the next Chapter the existing literature on models to represent probabilistic behaviour. We describe thereafter Probabilistic Component Automata, our probabilistic modelling formalism.

3. Modelling Probabilistic Behaviour and Reliability

Automating architectural reconfiguration requires a close integration between composable models of architectural, behaviour, non-functional properties and management aspects. An essential aspect is the *composability* of the models i.e., the ability to derive the representation of composite components from the representations of their parts. Non-functional properties such as reliability require extending behaviour models with stochastic information on the frequency of execution of actions. However, existing probabilistic models do not support composability as composing stochastic information is a challenging task [SV04]. To this end, we present in this Chapter Probabilistic Component Automata (PCA), a composable formalism to model the probabilistic behaviour of component-based systems. We extend Interface Automata with probabilities and additional constructs for failure modelling, propagation and handling to support automated analysis of reliability properties. The semantics of PCA models are intuitive and closely resemble the behaviour of component-based applications. In addition, by establishing a close link with architectural models, such as Darwin, we are able to automatically construct the composite the composite representation of different architectural configurations. For each architectural configuration, we modify the generic PCA representation of each component prior to composition to reflect the bindings with other components. Reliability requirements of a given system configuration are then automatically analysed using the corresponding composite PCA model.

Before we describe in Section 3.3 Probabilistic Component Automata in detail, we start by discussing in the next Section the differences between performance models and models of probabilistic choices in terms of their semantics and the properties that can be analysed using each type of model. We review thereafter the existing work on reliability analysis using models of probabilistic choices.

3.1. Comparison between CTMC and DTMC models

Discrete-Time Markov Chains (DTMC) and Continuous-time Markov Chains (CTMC) are both probabilistic models that support stochastic model checking [KNP07]. While in a DTMC model probabilistic choices are associated with single discrete steps between states, a CTMC model represents the rates of transitions from one state to another. Probabilistic choice, in CTMC models, occurs through race conditions when two or more transitions in a state are enabled. In the next paragraphs we discuss the differences between these formalisms, as they have both been used for reliability analysis [ST07, FGT12].

A DMTC is defined by a tuple $\langle \mathcal{S}, q, P, L \rangle$ where

- $\mathcal{S}$ is a finite set of states;
- $q \in \mathcal{S}$ is the initial state;
- $P : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ is the transition probability matrix, where $\forall s \in \mathcal{S} \sum_{s' \in \mathcal{S}} P(s, s') = 1$;
- $L : \mathcal{S} \rightarrow 2^{AP}$ is a labelling function that assigns to each state $s \in \mathcal{S}$ the set $L(s)$ of atomic propositions that are valid in that state.

Cheung [Che80] proposed using DTMC models to calculate the reliability of a system based on the reliability of its components. In such representations the matrix P models the transfer of control between components as follows: a label is assigned to each state s_i to denote that component C_i has the control over the execution of the system and $P(s_i, s_j)$ represents a transfer of control between component C_i and component C_j . If component C_i fails during its execution, a transition to a global failure F absorbing state, *i.e.* without transitions to other states, is added to state s_i , while $P(s_i, F)$ denotes the probability of such failure happening. A final absorbing state C corresponds to the correct execution of the system.

Consider the e-commerce system used by Filieri *et. al* [FGT12] which consists of a web-service that sells merchandise and integrates three external web-services: authentication, shipping and payment. A high-level description of the behaviour of the system is depicted by the activity diagram of Figure 3.1. The corresponding DTMC representation of the e-commerce system is shown in Figure 3.2 [FGT12]. Following Cheung's approach, each state is labelled with either an internal action of the e-commerce system or with the name of an external service-based, *e.g.* ExpShipping, while the probabilities of transitions between states model the usage profile as well as the probability of failures. For instance, 35% of the users are returning customers while

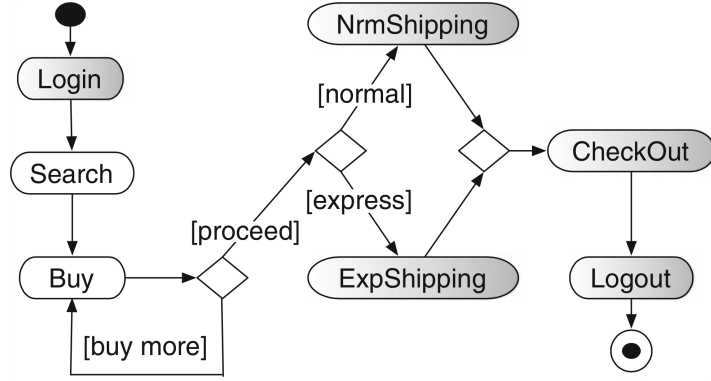


Figure 3.1.: Activity Diagram of an e-commerce System [FGT12]

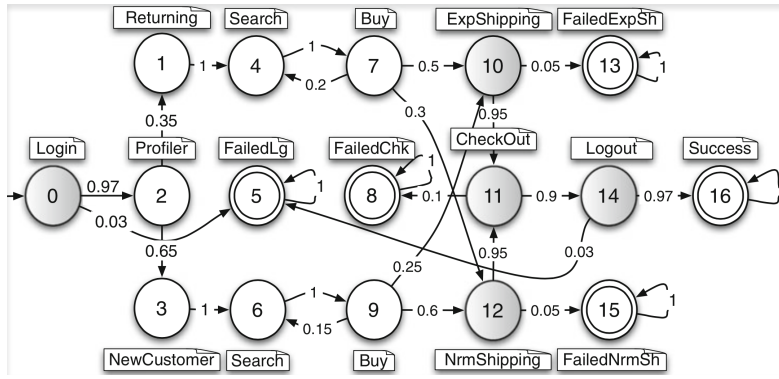


Figure 3.2.: DTMC Model of e-commerce System [FGT12]

the ExpShipping service has 5% probability of failing. The following reliability properties can then be analysed using this model:

- Probability of a request finishing without failures, *i.e.* the probability of reaching *Success* state;
- Probability of a failure by the ExpShipping service for a returning customer;
- Probability of an authentication failure.

On the other hand, CTMC models are used to analyse performance metrics such as the mean time for the system to fail. A CTMC is defined by a tuple $\langle \mathcal{S}, q, R, L \rangle$ where

- $\mathcal{S}$ is a finite set of states;
- $q \in \mathcal{S}$ is the initial state;
- $R : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ is a rate matrix, where a transition can only occur between states s and s' if $R(s, s') > 0$;

- $L : \mathcal{S} \rightarrow 2^{AP}$ is a labelling function that assigns to each state $s \in \mathcal{S}$ the set $L(s)$ of atomic propositions that are valid in that state.

The matrix R specifies the rate at which a transition is made between states s and s' , which is derived from the duration of making such transition. The semantics of CTMC models stipulates that if a state has more than one outgoing transition, then a *race condition* occurs between the enabled transitions from the state. In other words, the transitions are considered to be running in parallel and the action that first finishes its execution determines the next state. Furthermore, the time spent in each state $s \in \mathcal{S}$ before making a transition is designated as its *exit rate* $E(s)$, which corresponds to the time taken by the *fastest* transition to complete. Given that all the outgoing transitions of a state $s \in \mathcal{S}$ are modelled using an exponentially distributed random variable $X_{s,s'}$ with parameter $\lambda_{s,s'} = R(s, s')$, the exit rate $E(s)$ is defined as the minimum of those exponential distributions: $\min \{X_{s,1}, \dots, X_{s,n}\}$, where n is the number of states in $\mathcal{S}$. The resulting distribution is also exponential whose rate is defined as $E(s) = \sum_{s' \in \mathcal{S}} R(s, s')$. Moreover, the properties of the exponential distribution also stipulate that $P(X_{s,s'} = \min \{X_{s,1}, \dots, X_{s,n}\}) = \frac{\lambda_{s,s'}}{\lambda_{s,1} + \dots + \lambda_{s,n}}$. Therefore, the probability of a transition from state s to state s' being triggered before the others is given by $\frac{R(s,s')}{E(s)}$.

An embedded DTMC model is defined for a CTMC D model defined by tuple $\langle \mathcal{S}, q, R, L \rangle$ as $D = \text{emb}(C) = \langle \mathcal{S}, q, P, L \rangle$, where for each $s, s' \in \mathcal{S}$:

$$P^{\text{emb}(C)}(s, s') = \begin{cases} \frac{R(s,s')}{E(s)} & \text{if } E(s) \neq 0; \\ 1 & \text{if } E(s) = 0 \text{ and } s = s'; \\ 0 & \text{otherwise.} \end{cases}$$

Note that the embedded DTMC $\text{emb}(C)$ stipulates, for each state s , the probability that a transition to a state s' finishes before the others. **If the transitions from a state s represent the execution of multiple actions then the probability defined by the embedded DTMC is different than the probability represented in the DTMC associated with the usage profile.** For instance, the system can choose an action a with 90% and an action b with 10%, regardless of the time they require to complete. If action b takes significantly less time to complete than action a , then the embedded DTMC would define a significantly lower probability for action a . On the other hand, if the transitions from state s denote multiple outcomes of the same action a , the probabilities defined by $P^{\text{emb}(C)}(s, s')$ denote the probabilistic

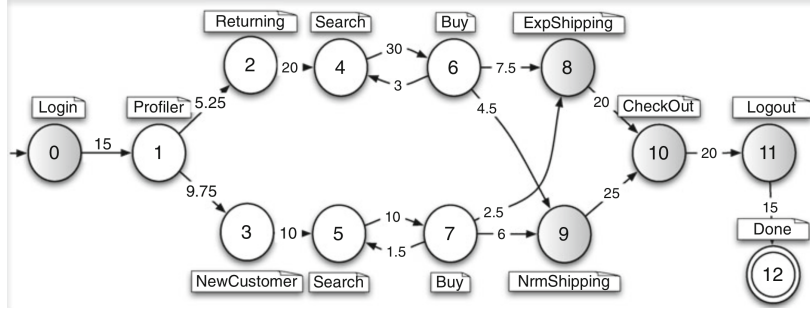


Figure 3.3.: CTMC Model of e-commerce System [FGT12]

choice over those outcomes. Given the rate a_r associated with an action a , consider two possible outcomes with probability p and $1 - p$. The rate of each transition can be defined as $p \cdot a_r$, the exit rate of state s $E(s) = p \cdot a_r + (1 - p) \cdot a_r$ denotes the rate of the action a executing in state s . In this special cases, $\frac{R(s, s')}{E(s)}$ designates the probability of each outcome of action a and a CTMC model can be used in a similar way to a DTMC model for reliability analysis based on the frequency of execution of failure and non-failure actions.

Consider now the CTMC model of the e-commerce system in Figure 3.3, whose rate matrix R is obtained by combining the rate of each action (in requests per second) with the usage profile defined by the DTMC model of the e-commerce in Figure 3.2. In other words, this representation is constructed by considering the DTMC model shown in Figure 3.2 as the embedded DTMC and by computing the rate of each transition $R(s, s')$ using the following formula :

$$R(s, s') = E(s) \cdot P^{emb(C)}.$$

Note that the CTMC model does not support analysis involving failure scenarios as it does not include transitions to failure states. Although the same procedure that was applied to transitions denoting successful execution could be applied to transitions to failure states, the use of the same rate for successful execution and failure scenarios implies that the amount of time to complete the full execution of a service and the detection of its failure would be the same. The CTMC model depicted in Figure 3.3 enables the analysis of the following properties:

- Probability that a session (from Login to Logout) requires at most x seconds to complete;
- Probability that the time needed for a returning customer until he/she checks out is less than x seconds;

- Probability that a new customer completes a full session using NormalShipping service waiting no more than x seconds.

Therefore, the properties that can be analysed in CTMC models are different from those that can be analysed in DTMC models. *In the latter, analysis is concerned with the probability of reaching a given state regardless of the time it takes, whereas in the former analysis is concerned with throughput, utilisation, mean time to reach a state or probability to reach within x units of time.* In the rest of this Chapter we focus on models that represent probabilistic choice as our focus is on reliability analysis, as previously described for DTMC models. We refer to Kwiatkowska *et al.* work on Stochastic Model Checking [KNP07] for a more detailed comparison between DTMC and CTMC models.

3.2. Models of probabilistic choice

Although DTMC models support automated analysis of reliability properties, a separate DTMC representation is required to be specified for each architectural configuration as these representations cannot be automatically constructed from a DTMC model of each component. For example, DTMC representations that follow Cheung’s approach [Che80] do not include the behaviour of a component, as each state only represents the execution of a single component. Therefore, when different architectural configurations are considered, a new representation has to be manually defined and the system needs to be profiled again to obtain a new transition matrix P .

Even if the information in the system model is extracted from DTMC representations of each component, there is no method to automatically construct the system model from those individual DTMC representations. Although different extensions to Cheung’s work have been proposed in an attempt to solve this limitation, they resort to manual specifications to *link* different DTMC models and construct a DTMC of the system. For instance, Wang *et. al* [WPC06] defined mappings between architectural patterns and DTMC representations of the entire system, such mappings must be manually defined and the resulting models do not include the internal behaviour of each component. On the other hand, Eteessami *et al.* [EY09] extended DTMC models with the notion of input and output states in order to model the behaviour of each component as an individual DMTC. The model of the system is then constructed by *merging* input-output states of DTMC models. Similarly to Wang *et. al* work, the links between

input-output states are defined manually and the method does not overcome the main problems associated with DTMC models, *e.g.* only sequential execution is modelled.

We illustrate these limitations in the next paragraphs using a series of small examples. Consider the DTMC models A and B in figures 3.4(a) and 3.4(b), respectively. They both make a choice from their initial state that leads to two other states (outcomes). When these two DTMC models are composed, as shown in Figure 3.6, each step in the composed DTMC consists of joint independent steps performed by the two components, as the parallel composition of two DTMCs is synchronous.

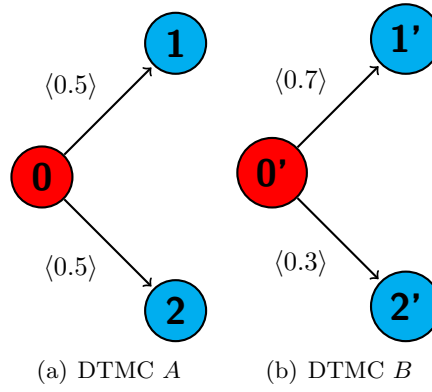


Figure 3.4.: Examples of DTMC Models

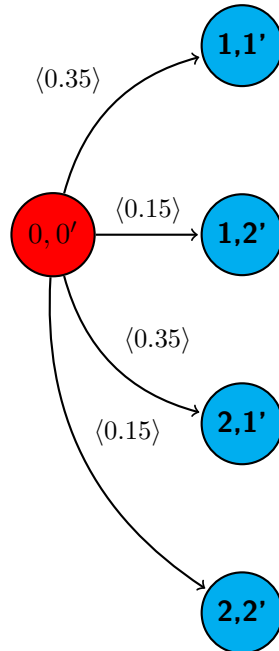


Figure 3.5.: Synchronous Composition of DTMC Models A and B

Consider now an alternative composition which aims at representing the concurrent execution of A and B . The composite model in Figure 3.6 does not preserve the semantics of generative systems as its transitions from composite state $\langle 0, 0' \rangle$ do not sum up to 1. Intuitively, the DTMC A is making a probabilistic choice in its state 0 which is independent from the probabilistic choice made by B in state $0'$. How should the probabilities from composite state $\langle 0, 0' \rangle$ be changed such that their sum equals 1? *Normalising the probabilities for their sum is not always desirable and valid as modelling concurrent execution of independent choices presents several challenges [SV04].*

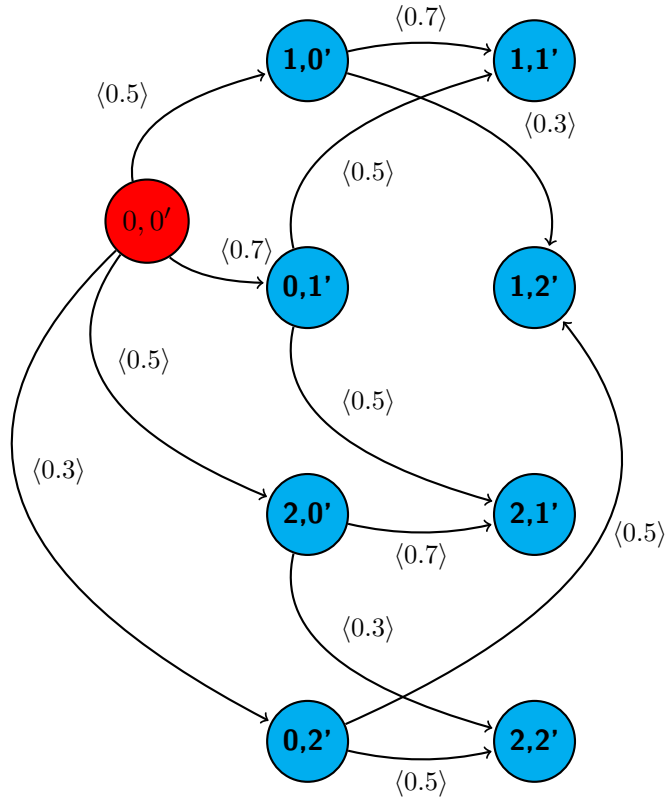


Figure 3.6.: Composition of DTMC Models A and B

Furthermore, DTMC-based approaches assume that failures occur independently in components bound to each other and cannot represent failure dependencies and failure propagation across component bindings. Filieri *et. al* [FGGM10] extended Cheung's approach to enable modelling of failure propagation but their approach still does not support automatic construction of the model from the representations of its parts.

Alternatively, *path-based* approaches consider the reliability of components and communication channels. The behaviour of a system is modelled as disjoint subdomains to represent the

system operational profile [KM97]. System reliability is then calculated from the reliability of each subdomain. Yacout *et al.* [YCA04] represent subdomains (execution paths) using Message Sequence Charts that are subsequently translated to a Component Dependency Graph (CDG) under the assumption that the system architecture is hierarchical. The CDG begins with a start node from which the execution paths are initiated and terminates with an end node, which denotes the completion of all the paths. System reliability is calculated using a breadth-first search on the CDG as follows: at each step, a component is visited by the algorithm and the system reliability is calculated as

$$R = R_{previous} \times R_{C_i} \times R_{T_{ij}} \times P_{T_{ij}},$$

where $R_{previous}$ is the reliability of the components already *visited*, R_{C_i} is the reliability associated with component C_i , $R_{T_{ij}}$ is the reliability associated with the transfer of control between C_i and C_j , and $P_{T_{ij}}$ is the probability of control being transferred from C_i to C_j . As loops may be present in the CDG, each component is associated with an expected execution time and an upper bound limit on the system execution time is specified to avoid infinite iterations of the graph navigation algorithm. When this limit is reached, the reliability computation is stopped. Furthermore, a separate CDG can be used to encode the behaviour of a composite component [YCA04]. Zhang *et al.* [ZZCD08] extended the previous CDG-based model by applying the subdomains approach to each component to represent its operational profile, thus enabling more fine-grained modelling and analysis.

However, path-based models encompass similar limitations to the ones described for DTMC models, *e.g.* failures are assumed to be independent. Although the system model proposed by Cheung *et al.* [CRMG08] is computed based on the representations of its components with support for the representation of internal behaviour, these representations are specific to the context in which the components are deployed, hence they cannot be directly used in other systems. One of the reasons for this limitation arises from the requirement of having fully probabilistic models, where deterministic choices are made at each state. However, some aspects of a system may not be probabilistic or cannot be modelled as a probabilistic choice, *e.g.* unknown execution environment. Markov Decision Processes [DA98] are an extension of DTMC models that allow for the representation of non-deterministic choices. In each state, a non-deterministic choice between several discrete probability distributions over successor states is made. Although MDP models support analysis of worst and best-case scenarios, exact probabilistic analysis requires

non-deterministic choices to be resolved first. A single or multiple adversaries, also known as *scheduler* or *policy*, resolve(s) non-deterministic choices by specifying probabilities over paths, thereby producing a DTMC representation. However, MDPs also present limitations when constructing composite representations from an MDP model of each component. For instance, if two MDPs make independent deterministic choices from states s and s' , when composing these models a non-deterministic choice is specified in for the composite state $\langle s, s' \rangle$, which needs to be resolved later by an adversary. In addition, when two MDPs are composed, the deterministic choices associated with required interfaces of a component are *synchronised* with the non-deterministic choices associated provided interfaces bound to those required interfaces. Although a component without provided interfaces can be modelled as an adversary, it cannot resolve all the probabilistic choices expressed by bindings in lower levels of the composition hierarchy.

In the next Section we discuss other approaches to compose probabilistic models.

3.2.1. Composable Probabilistic Models

Several probabilistic extensions have been proposed for Labelled Transition Systems based on:

- adding probabilities to every transition or to transitions labelled with the same action;
- making a distinction between probabilistic and non-deterministic states, where probabilistic choices are only specified for the former;
- defining a transition function which supports both non-determinism and probabilistic choices.

The first type of extension are denoted by *generative* and *reactive* probabilistic systems. In generative systems a probability distribution is associated with all outgoing transitions from a given state, since actions denote probabilistic choices performed by the system, and their sum must equal 1. Formally, a generative PLTS model is defined by the tuple $A = \langle \mathcal{S}, q, \mathcal{E}, \mu \rangle$ where $\mu : \Delta \rightarrow [0, 1]$ assigns a probability to each transition in Δ such that

$$\forall s \in \mathcal{S}, \quad \sum_{(s, a, s') \in \Delta} \mu(s, a, s') = 1.$$

When two generative systems are composed in parallel, it is not clear how the probabilities of transitions in the composite model should be defined [SV04]. In the next paragraphs we

illustrate different cases of parallel composition for composition the generative models in Figure 3.7.

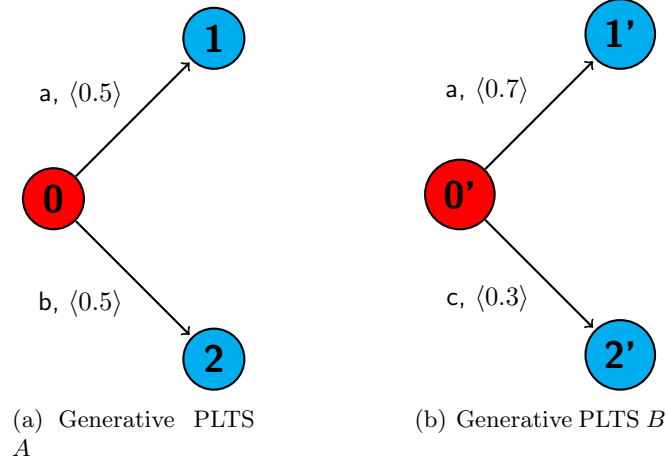


Figure 3.7.: Examples of Generative PLTS Models

Given a composite state $\langle s, t \rangle$, a transition from state s to state s' labelled with action a and with probability p_a ($s \xrightarrow{a, p_a} s'$) from one generative PLTS model, and a transition from state t to state t' labelled with action b and with probability p_b ($t \xrightarrow{b, p_b} t'$) from another generative PLTS model, synchronous parallel composition semantics defines that the transitions are *synchronised* in the composite model as transition $\langle s, t \rangle \xrightarrow{ab, p_a \cdot p_b} \langle s', t' \rangle$. The corresponding composite generative PLTS using synchronous composition is shown in Figure 3.8.

The synchronous composition rule preserves the semantics of generative systems for this composite model. However, if the rule is applied to the parallel composition between the generative PLTS model A in Figure 3.7(a) and the generative PLTS model C in Figure 3.9, the resulting composite model does not preserve the generative semantics of each state, as the sum of the probabilities of outgoing transitions from the initial composite state $\langle 0, 0'' \rangle$ in the corresponding composite model show in Figure 3.10 does not sum to 1. In addition, the synchronous parallel composition rules do not capture non-determinism associated with concurrent execution of non-shared actions.

A parameterised parallel composition operator $\parallel_{\sigma, \theta}$ is proposed by Baeten *et al.* [BBS95] which considers both synchronous and asynchronous transitions. When composing two generative PLTS models A and B , the two parameters σ and θ are applied as follows. Consider the composite state $\langle s, t \rangle$; transitions from state s and state t are synchronised with probability $1 - \theta$ and asynchronous transitions occur with probability θ ; an asynchronous move is performed

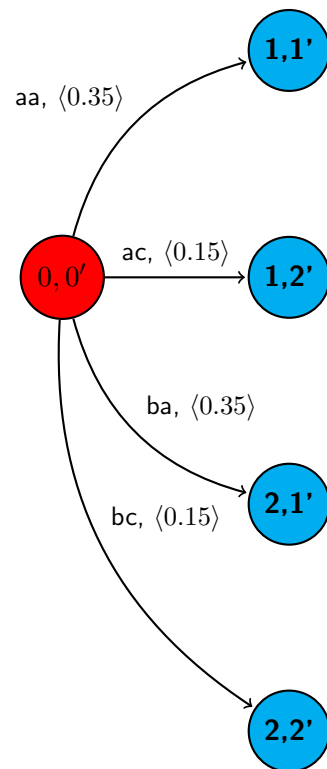


Figure 3.8.: Synchronous Composition of Generative PLTS Models A and B

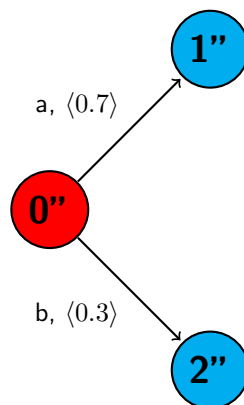
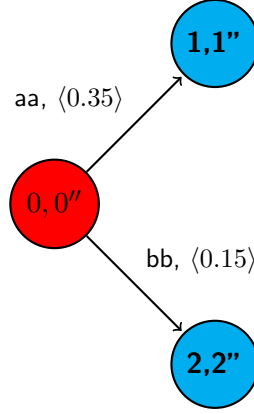


Figure 3.9.: Generative PLTS Model C


 Figure 3.10.: Synchronous Composition of Generative PLTS Models A and C

by model A with probability σ and by model B with probability $1 - \sigma$. If only either A or B can make a transition, the parameters σ and θ are not applied.

Another parameterised parallel composition operator $\parallel_L^\sigma$ is defined by D'Argenio *et al.* [DHK99], where synchronisation solely occurs between transitions with shared actions in the set of action labels L . The factor σ denotes the probability that model A performs autonomous actions, given that both A and B have *decided* not to synchronise at a given composite state $\langle s, t \rangle$. The following cases denote the situations where a transition $\langle s, t \rangle \xrightarrow{a, P} \langle s', t' \rangle$ is included in the composite model:

- A makes an autonomous move, when B can also do an autonomous move:

$$s \xrightarrow{a, p} s', t \xrightarrow{b, q} t', a, b \notin L, t' = t \text{ then } P = \frac{p \cdot q \cdot \sigma}{\nu(s, t, L)};$$

- B makes an autonomous move, when A can also do an autonomous move:

$$s \xrightarrow{a, p} s', t \xrightarrow{b, q} t', a, b \notin L, s' = s \text{ then } P = \frac{p \cdot q \cdot \sigma}{\nu(s, t, L)};$$

- A makes an autonomous move, but B blocks as it cannot synchronise:

$$s \xrightarrow{a, p} s', t \xrightarrow{b, q} t', a \notin L, b \in L, t' = t \text{ then } P = \frac{pq}{\nu(s, t, L)};$$

- B makes an autonomous move, but A blocks as it cannot synchronise:

$$s \xrightarrow{b, p} s', t \xrightarrow{a, q} t', a \notin L, b \in L, s' = s \text{ then } P = \frac{pq}{\nu(s, t, L)};$$

- A and B make a synchronous move:

$$s \xrightarrow{a,p} s', t \xrightarrow{a,q} t', a \in L, \text{ then } P = \frac{pq}{\nu(s, t, L)};$$

- A makes an autonomous move, but B is in a deadlock state:

$$s \xrightarrow{a,p} s', a \notin L, t' = t \text{ then } P = \frac{p}{\nu'(s, L)};$$

- B makes an autonomous move, but A is in a deadlock state:

$$t \xrightarrow{b,q} t', b \notin L, s' = s \text{ then } P = \frac{q}{\nu'(t, L)};$$

ν and ν' are normalisation factors defined as follows:

$$\nu(s, t, L) = 1 - \sum_{\substack{s \xrightarrow{a,p} s' \\ t \xrightarrow{b,q} t' \\ a, b \in L \\ a \neq b}} pq,$$

$$\nu'(s, L) = 1 - \sum_{\substack{s \xrightarrow{a,p} s' \\ a \in L}} p.$$

Both normalisation factors distribute the probability of transitions that are blocked to the transitions included in the composite model to ensure the semantics of generative systems are preserved.

Although all the previous parallel composition operators enable the construction of composite models that comply with the semantics of generative systems. Each individual model *can only represent probabilistic choices* that a component makes on its internal behaviour and on invocations to other components through required interfaces, as generative models *cannot cater for non-deterministic choices* associated with provided interfaces.

In contrast, reactive systems associate a discrete probability distribution with the outgoing transitions of a state labelled with the same action. Therefore, actions are treated as input from the environment, *i.e.* reactive PLTS only cater for non-deterministic choices as a model does not decide which action is chosen at each state but only their possible outcomes. Formally, a reactive PLTS model is defined by the tuple $A = \langle \mathcal{S}, q, \mathcal{E}, \mu \rangle$ where $\mu : (s, a) \rightarrow \mu_{s,a}$ assigns a

probabilistic distribution $\mu_{s,a}$ to state $s \in \mathcal{S}$ and action label $a \in \mathcal{E}$, which in turn defines the outgoing states and the associated probabilities $\mu_{s,a} : s' \rightarrow [0, 1], s' \in \mathcal{S}$ such that

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{E}, \text{ if } \mu_{s,a} \in \mu, \sum_{s' \in \mathcal{S}} \mu_{s,a}(s') = 1.$$

Given two reactive PLTS models A and B , their composition is specified by $A \parallel B = \langle \mathcal{S}_A \times \mathcal{S}_B, \langle q_A, q_B \rangle, \mathcal{E}_A \cup \mathcal{E}_B, \mu_{AB} \rangle$. For each composite state $\langle s, t \rangle$ and $a \in \mathcal{E}_{AB}$, $\mu_{\langle s, t \rangle, a}$ is defined as follows:

- if $a \in \mathcal{E}_A \cap \mathcal{E}_B$, then $\mu_{\langle s, t \rangle, a} : \langle s', t' \rangle \rightarrow \mu_{s,a}(s') \times \mu_{t,a}(t')$;
- if $a \in \mathcal{E}_A \wedge a \notin \mathcal{E}_B$, then $\mu_{\langle s, t \rangle, a} : \langle s', t \rangle \rightarrow \mu_{s,a}(s') \times 1$;
- if $a \in \mathcal{E}_B \wedge a \notin \mathcal{E}_A$, then $\mu_{\langle s, t \rangle, a} : \langle s, t' \rangle \rightarrow \mu_{t,a}(t') \times 1$.

Although the composition of reactive PLTS models does not require normalisation of probabilities in the composite model, such representation does not enable exact probabilistic analysis as non-deterministic choices performed by the environment need to be resolved. To this end, Probabilistic Interface Automata (PIA) [PBU] introduce the composition of a probabilistic model of the environment with a non-probabilistic model of the software system, which combines Interface Automata with Markov Decision Processes. Although PIA can be composed, choices are only resolved by a single probabilistic representation of the environment, similar to the scheduler previously defined for MDP representations.

Alternatively, Probabilistic I/O Automata (PIOA) [WSS94] provide a hybrid solution by combining reactive and generative systems. This is achieved by extending I/O Automata [LT87] with probabilities and associating reactive semantics with input actions and generative semantics with internal and output actions. A PIOA model is formally defined by the tuple $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu, \delta \rangle$, where

- $\mathcal{S}$ is a finite set of states, $q \in \mathcal{S}$ is the initial state;
- $\mathcal{E} = \mathcal{E}^{in} \cup \mathcal{E}^{out} \cup \mathcal{E}^{int}$ is a set of action labels comprise of a set of input actions $\mathcal{E}^{in}$, a set of output actions $\mathcal{E}^{out}$ and a set of internal actions $\mathcal{E}^{int}$;
- $\Delta \subseteq (\mathcal{S} \times \mathcal{E} \times \mathcal{S})$ is the set of transitions;

- $\mu : \Delta \rightarrow [0, 1]$ is the transition function subject to the following rules:

$$\underbrace{\forall s \in \mathcal{S}, \left(\sum_{\substack{(s,a,s') \in \Delta \\ a \in \mathcal{E}^{loc}}} \mu(s, a, s') \right) = 1}_{\text{generative semantics}} \quad (3.1)$$

$$\underbrace{\forall s \in \mathcal{S}, \forall a \in \mathcal{E}^{in}, \left(\sum_{(s,a,s') \in \Delta} \mu(s, a, s') \right) = 1}_{\text{reactive semantics}} \quad (3.2)$$

- $\delta : \mathcal{S} \rightarrow [0, \infty)$ is a state delay function which assigns a delay rate to each state.

When composing two PIOA models $A = \langle \mathcal{S}_A, q_A, \mathcal{E}_A, \Delta_A, \mu_A, \delta_A \rangle$ and $B = \langle \mathcal{S}_B, q_B, \mathcal{E}_B, \Delta_B, \mu_B, \delta_B \rangle$ the following cases are considered w.r.t. synchronisation of interface actions:

- synchronisation can occur between input actions, as in reactive systems;
- synchronisation involving output actions is only allowed for matching pairs of input-output actions.

Therefore, parallel composition is only defined for *compatible* PIOA models, with disjoint sets of output actions ($\mathcal{E}_A^{out} \cap \mathcal{E}_B^{out} = \emptyset$) and distinguishable internal actions such that $\mathcal{E}_A^{int} \cap \mathcal{E}_B = \emptyset$ and $\mathcal{E}_B^{int} \cap \mathcal{E}_A = \emptyset$. The composite PIOA $AB = \langle \mathcal{S}_A \times \mathcal{S}_B, \langle q_A, q_B \rangle, \mathcal{E}_{AB}, \Delta_{AB}, \mu_{AB}, \delta_{AB} \rangle$ is defined as follows:

- $\mathcal{S}_{AB} = \mathcal{S}_A \times \mathcal{S}_B$;
- $q_{AB} = \langle q_A, q_B \rangle$;
- $\mathcal{E}_{AB} = \mathcal{E}_{AB}^{in} \cup \mathcal{E}_{AB}^{out} \cup \mathcal{E}_{AB}^{int}$, where
 1. $\mathcal{E}_{AB}^{in} = (\mathcal{E}_A^{in} \cup \mathcal{E}_B^{in}) \setminus (\mathcal{E}_A^{out} \cup \mathcal{E}_B^{out})$;
 2. $\mathcal{E}_{AB}^{int} = \mathcal{E}_A^{int} \cup \mathcal{E}_B^{int} \cup (\mathcal{E}_A^{in} \cap \mathcal{E}_B^{out}) \cup (\mathcal{E}_B^{in} \cap \mathcal{E}_A^{out})$;
 3. $\mathcal{E}_{AB}^{out} = (\mathcal{E}_A^{out} \cup \mathcal{E}_B^{out}) \setminus (\mathcal{E}_A^{in} \cup \mathcal{E}_B^{in})$;
- $\Delta_{AB} = \text{set of all } \langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle \text{ such that}$
 1. $s_A \xrightarrow{a} s'_A \in \Delta_A$ and $s_B \xrightarrow{a} s'_B \in \Delta_B$;
 2. $s_A \xrightarrow{a} s'_A \in \Delta_A$ and $s_B \xrightarrow{a} s'_B \notin \Delta_B \implies s_B = s'_B$;

$$3. s_B \xrightarrow{a} s'_B \in \Delta_B \text{ and } s_A \xrightarrow{a} s'_A \notin \Delta_A \implies s_A = s'_A;$$

$$\bullet \forall \langle s_A, s_B \rangle \in \mathcal{S}_{AB}, \quad \delta_{AB}(\langle s_A, s_B \rangle) = \delta_A(s_A) + \delta_B(s_B)$$

$$\bullet \forall \langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle \in \Delta_{AB}$$

$$1. \text{ if } a \in \mathcal{E}_{AB}^{in},$$

$$\mu_{AB}(\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle) = \mu_A(s_A, a, s'_A) \cdot \mu_B(s_B, a, s'_B);$$

$$2. \text{ if } a \in \mathcal{E}_A^{out} \wedge a \in \mathcal{E}_B^{in},$$

$$\mu_{AB}(\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle) = \frac{\delta_A(s_A)}{\delta_A(s_A) + \delta_B(s_B)} \cdot \mu_A(s_A, a, s'_A) \cdot \mu_B(s_B, a, s'_B);$$

$$3. \text{ if } a \in \mathcal{E}_B^{out} \wedge a \in \mathcal{E}_A^{in},$$

$$\mu_{AB}(\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle) = \frac{\delta_B(s_B)}{\delta_A(s_A) + \delta_B(s_B)} \cdot \mu_A(s_A, a, s'_A) \cdot \mu_B(s_B, a, s'_B);$$

$$4. \text{ if } a \in \mathcal{E}_A^{loc} \wedge a \notin \mathcal{E}_B^{in},$$

$$\mu_{AB}(\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle) = \frac{\delta_A(s_A)}{\delta_A(s_A) + \delta_B(s_B)} \cdot \mu_A(s_A, a, s'_A);$$

$$5. \text{ if } a \in \mathcal{E}_B^{loc} \wedge a \notin \mathcal{E}_A^{in},$$

$$\mu_{AB}(\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle) = \frac{\delta_B(s_B)}{\delta_A(s_A) + \delta_B(s_B)} \cdot \mu_B(s_B, a, s'_B);$$

Given that synchronisation occurs only between matching pairs of input-output actions, which can correspond to the bindings between the *provided* and *required* interfaces of components, or pairs of input-input actions, PIOA A and B perform a synchronous move from a composite state $\langle s_A, s_B \rangle$ if they can both perform a matching action a . The synchronisation of matching pairs of input-output actions results in a single transition labelled with an internal actions in the composite PIOA. Rules 4 and 5 for Δ_{AB} denote asynchronous execution of internal actions or output actions of PIOA A (B) for which there is no corresponding input action in PIOA B (A). The probability of composite transitions, as established by μ_{AB} , is described as follows. The first rule denotes the synchronisation of input actions as defined for reactive systems. Rules 2 and 3 denote the probability of synchronised input-output pairs of actions where deterministic choices performed by output actions are combined with non-deterministic choices associated with matching input actions, while rules 4 and 5 specify the probability for internal and output when they execute asynchronously. In both cases the state delay rate from each PIOA is used to

normalise the transitions, thereby ensuring the semantics of generative systems are preserved. The normalisation rule is underpinned by a race condition between the two PIOA, equivalent to its use for CTMC models.

Although PIOA models overcome some of the problems of the models previously discussed by combining the semantics of reactive and generative systems, the *input-enabledness* requirement presents several difficulties when modelling the behaviour of software components. Consider the case of modelling a single-threaded server component using a PIOA representation. The semantics of PIOA models implies that the server must process requests from clients (input actions) at any time, even when the server is internally handling a request. Input-enabledness also does not support modelling of assumptions regarding interactions with the environment. For example, when modelling a particular communication/interaction algorithm, *e.g.* TCP handshake, each side assumes a particular order of interaction.

Alternatively Probabilistic Component Interface Protocols (PCIP) [KGM10] combine elements of PIOA and IA to specify representations that do not require input-enabledness. A PCIP model is defined by the tuple $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu, \delta \rangle$, and the construction of composite models is similar to the rules previously described for PIOA models. Synchronisation is only applied to matching pairs of input-output actions and delay rates are used for normalisation of transitions that follow generative semantics. On the other hand, while input actions block until they can be synchronised with the corresponding output action, as specified for IA models, the execution of output actions is never blocked. When composing two PCIP models A and B , for a given composite state $\langle s_A, s_B \rangle$, if A is ready to execute an output action¹, *i.e.* $s_A \xrightarrow{a} s'_A \in \Delta_A \wedge a \in \mathcal{E}_A^{out}$, then the following two cases are considered by the parallel composition operator of PCIP:

- if B is ready to execute the corresponding input action, $s_B \xrightarrow{a} s'_B \in \Delta_B \wedge a \in \mathcal{E}_B^{in}$, then A and B interact synchronously and the transition $\langle s_A, s_B \rangle \xrightarrow{a} \langle s'_A, s'_B \rangle$ is added to the composite model, following synchronisation semantics of IA and PIOA;
- if B cannot execute the corresponding input action from state s_B , then a transition $\langle s_A, s_B \rangle \xrightarrow{a} \langle E_A, s_B \rangle$ is added to the composite model leading to an error state $\langle E_A, s_B \rangle$.

In contrast with IA models where output actions are blocked until a corresponding input action is available for synchronisation, PCIP models consider blocking of output actions inconsistent behaviour and model such scenario as erroneous transitions to a special error state. This synchronisation semantics makes PCIP less suitable for reliability analysis where transitions to the

¹The case where B executes the output action is equivalently defined.

error state represent failures of actions, *e.g.* communication failures. Note also that analysis of liveness properties can be used to verify if a given output action is always executed, without having to explicitly represent in the model the situations where the execution of an output action has been blocked.

Although input/output actions can be associated with provided/required interfaces, none of the described models establishes a link between behaviour and architectural models. In fact, when constructing the composite model for a given composition of components, these models implicitly assume a configuration with single bindings between all provided and required interfaces. How can one construct the composite representation for a configuration with unused functionality? What changes need to be made to the model of a component to enable the representation of multiple bindings to its provided interfaces? None of the previous models provides the means to cater for these scenarios. In addition, these models normalise the probabilities of concurrently executed actions using delay rates whose semantics is not well defined (see detailed discussion on Section 3.3.4). We will discuss the normalisation applied by both PCIP and PIOA in more detail in Section 3.3.2. Furthermore, failure scenarios are represented in the previous models using transitions labelled with internal actions leading to *failure* state². However, given that failure actions are not explicitly represented, these models do not cater for failure handling behaviour which requires a compositional semantics for failure actions.

3.3. Probabilistic Component Automata

Composable models of non-functional properties are required for autonomous systems to be able to automatically analyse the non-functional properties of a system configuration by deriving its representation from the (composite) models of its components. In this Section we propose our composable modelling formalism, Probabilistic Component Automata (PCA), as a probabilistic extension to Interface Automata, *i.e.* we follow the synchronisation semantics of IA and extend it with probabilistic information. We describe the main operators to model PCA models for basic components and their corresponding semantics. We extend the parallel composition operator rules of Interface Automata to consider probabilistic information associated with transitions, including rules for normalisation of concurrent execution of probabilistic choices. In contrast with the parallel composition algorithm for IA models which requires the construction of the

²Other approaches to reliability analysis based on the probability of reaching an *error state* as a result of failures are described in existing surveys [Gok07, KEC⁺08].

full product automaton, the algorithm we implement for composing PCA models is an extension of the parallel compositional algorithm for LTS models. As a result, the composite model is constructed iteratively without having to explore all possible composite states. Finally, we describe re-labelling and hiding operators which, in conjunction with the parallel composition operator, are integrated with the architectural models to enable automatic construction of the composite representation for a given system configuration.

A Probabilistic Component Automaton is formally defined by the tuple $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ where:

- $\mathcal{S}$ is a finite set of states and $q \in \mathcal{S}$ is the initial state;
- $\mathcal{E} = \mathcal{E}^{in} \cup \mathcal{E}^{loc}$:
 - $\mathcal{E}^{in}$ are input actions from the environment that follow reactive semantics;
 - $\mathcal{E}^{loc} = \mathcal{E}^{int} \cup \mathcal{E}^{out}$ are *locally controlled* actions that follow generative semantics, where
 - * $\mathcal{E}^{int}$ are internal actions and
 - * $\mathcal{E}^{out}$ are output actions.
- $\Delta \subseteq (\mathcal{S} \times \mathcal{E} \times \mathcal{S})$ is the set of transitions;
- $\mu : \Delta \rightarrow [0, 1]$ where $\mu(s, a, s')$ denotes the probability of reaching state s' from state s through the execution of action a , subject to:

$$\forall s \in \mathcal{S}, \underbrace{\left(\sum_{\substack{(s,a,s') \in \Delta \\ a \in \mathcal{E}^{loc}}} \mu(s, a, s') \right)}_{\text{generative semantics}} = 1 \quad (3.3)$$

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{E}^{in}, \underbrace{\left(\sum_{(s,a,s') \in \Delta} \mu(s, a, s') \right)}_{\text{reactive semantics}} = 1 \quad (3.4)$$

To cater for automated reliability analysis of different architecture configurations, we establish a close link with architectural models, such as Darwin, when modelling basic components as well as when automatically constructing the composite representation of each configuration based on generic PCA representations of each component. Consider a simple Client-Server system based

on a Client Component (Figure 3.11) and a Server Component (Figure 3.12), described using Darwin's graphical and textual notations. The Client component has a required interface r and the Server component has a provided interface p .

```

component Client {
require r;
}

```

(a) Client - Darwin textual representation



(b) Client - Darwin graphical representation

Figure 3.11.: Darwin Representations of Client Component

```

component Server {
provide p;
}

```

(a) Server - Darwin textual representation



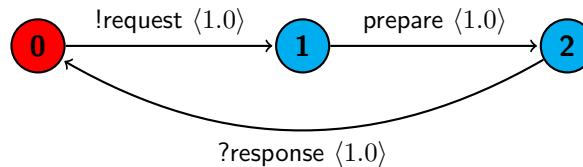
(b) Server - Darwin graphical representation

Figure 3.12.: Darwin Representations of Server Component

The corresponding PCA representations are given in Figures 3.13(b) and 3.14(b), respectively. Input-output actions are associated with the interfaces of a component, where the set $\{!request, ?response\}$ is associated with required interface r of the Client component and the set $\{?request, !response\}$ is associated with provided interface p of the Server component. In the general case for provided interfaces, an input action models the possibility of the associated method being invoked followed by internal behaviour and an output action denotes that the method result is being returned/sent. In the case of a required interface, an output action designates the invocation of the required functionality and an input action indicates the component receiving the result of the invoked functionality. The PCA representations in Figures 3.13(b) and 3.14(b) follow such structure.



(a) Client Component



(b) Client PCA

Figure 3.13.: Client Component and Associated PCA Representation

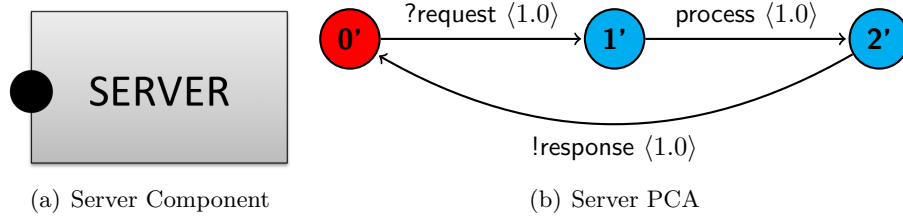


Figure 3.14.: Server Component and Associated PCA Representation

We introduce in the following Sections the different operators to construct probabilistic representations for basic and composite components.

3.3.1. Modelling Basic Components

PCA models are graphical representations that require the specification of a full model. Just as Finite State Processes are used as the textual notation for describing the behaviour of components which is then transformed in to LTS representations, we have defined Probabilistic Finite State Processes (P-FSP) (see Section 3.6) for incremental textual specification of PCA models. The correspondence between P-FSP expressions and PCA models is defined by the function $pca : E \rightarrow \text{PCA}$. Given a P-FSP expression E , $pca(E) = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$.

Prefix and choice are the basic operators to incrementally construct PCA models of basic components, whose operational semantics is respectively defined by Rule 1 and Rule 2³.

$$\frac{}{(a, p_a \rightarrow E) \xrightarrow{a, p_a} E} \quad (\text{Rule 1})$$

The prefix operator models the execution of a single statement represented as a transition, which includes:

- an action type: ? for input, ! for output, no-symbol for internal, $\sim$ for internal failures, $\sim?$ for input failures and $\sim!$ for output failures;
- the execution probability p , and
- the action label a .

The corresponding PCA model is given by $pca(a, p_a \rightarrow E) = \langle \mathcal{S} \cup \{p\}, p, \mathcal{E} \cup \{a\}, \Delta \cup \{(p, a, q)\}, \mu \cup \{(p, a, q) \rightarrow p_a\} \rangle$. Similarly to IA, input actions model the receiving end of a communication channel, *i.e.* (interface) methods that can be called, or any other form of input whose execution is

³The operational semantics rules should be read as $\frac{\text{Hypothesis}}{\text{Conclusion}}$.

determined by the environment, whereas output actions model the invocation of methods, sending of messages or any externally visible action whose execution is controlled by the PCA model. Internal actions denote internal behaviour that is not externally visible to other components.

$$\frac{}{(a_1, p_1 \rightarrow E_1 \mid \dots \mid a_n, p_n \rightarrow E_n) \xrightarrow{a_i, p_{a_i}} E_i} \quad (\text{Rule 2})$$

The *choice* operator enables the representation of different options from a given state whose semantics depend on the type of the actions $a_1 \dots a_n$. If $a_1, \dots, a_n$ are input actions with the same action label, as shown in Figure 3.15(a), then the choice operator is used as a syntactic simplification for the behaviour described by the PCA model in Figure 3.15(b): *after the environment decides to execute the output action $!a$ and synchronises with the corresponding input action $?a$, then the model performs a deterministic choice*. This structure can also be used to model multiple input values.

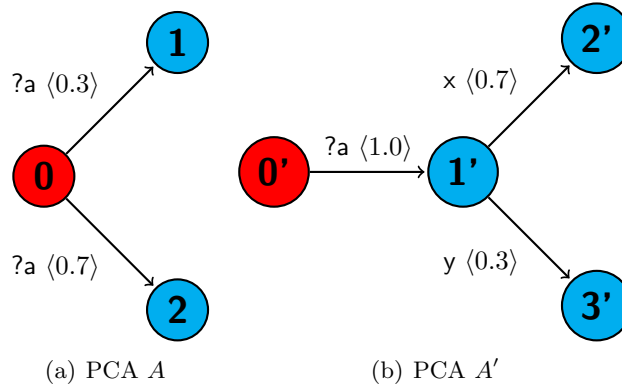


Figure 3.15.: Example Case for Choice Operator with Input Actions

The second case denotes a choice between input actions with different action labels corresponding to different requests from the environment. Each action can indicate different provided interfaces that can be invoked by the environment, or choices offered by a communication protocol.

Lastly, if $a_1, \dots, a_n$ are locally controlled actions (internal or output) of different action labels, then the choice operator models multiple outcomes of the same internal or output action, *i.e.* **if** or **switch** statements.

In the general case, the P-FSP expression $(a_1, p_1 \rightarrow E_1 \mid \dots \mid a_n, p_n \rightarrow E_n)$ describes a PCA that initially engages in any action a_i with probability p_i according to the previous

cases. The corresponding PCA model is formally defined by the following. Let $1 \leq j \leq n$ and $pca(E_j) = \langle \mathcal{S}_j, q_j, \mathcal{E}_j, \Delta_j, \mu_j \rangle$, then $pca((a_1, p_1 \rightarrow E_1 \mid \dots \mid a_n, p_n \rightarrow E_n \xrightarrow{a_i, p_i} E_i)) =$

$$\langle (\bigcup_i \mathcal{S}_i) \cup \{p\}, p, (\bigcup_i \mathcal{E}_i) \cup \{a_1, \dots, a_n\}, (\bigcup_i \Delta_i) \cup \{(p, a_i, q_i)\}, (\bigcup_i \mu_i) \cup \{(p, a_i, q_i) \rightarrow p_{a_i}\} \rangle.$$

3.3.2. Modelling Composite Components

While the previous operators enable the specification of basic components, the *parallel composition* operator $\parallel$ is used to automatically construct the PCA model of a composite component from the PCA models of its sub-components. The semantics of composite models is a probabilistic extension of the composition semantics of IA models: synchronisation occurs only between input and output actions to model the interactions between two components, while internal actions of different models are interleaved to denote their concurrent execution. Moreover, parallel composition can only be applied to *compatible* PCA models. Two PCA (A, B) are compatible iff:

$$\mathcal{E}_A^{int} \cap \mathcal{E}_B = \emptyset, \quad \mathcal{E}_B^{int} \cap \mathcal{E}_A = \emptyset,$$

$$\mathcal{E}_A^{in} \cap \mathcal{E}_B^{in} = \emptyset, \quad \mathcal{E}_A^{out} \cap \mathcal{E}_B^{out} = \emptyset.$$

These conditions ensure that synchronisation occurs solely between a single pair of input and output actions. In practice, the conditions imply that parallel composition can only be applied to synchronise single bindings to a provided interface. For configurations with multiple bindings, the interface actions of the components involved have to be differentiated before constructing the composite model, which we discuss later in Section 3.3.4.

$$\frac{A \xrightarrow{(!a, p_a)} A', B \xrightarrow{(?a, p_{a'})} B'}{A \parallel B \xrightarrow{(a, \frac{p_a \cdot p_{a'}}{\eta})} A' \parallel B'}, a \in \mathcal{E}_A^{out} \wedge \mathcal{E}_B^{in}$$

$$\frac{A \xrightarrow{(?a, p_a)} A', B \xrightarrow{(!a, p_{a'})} B'}{A \parallel B \xrightarrow{(a, \frac{p_a \cdot p_{a'}}{\eta})} A' \parallel B'}, a \in \mathcal{E}_A^{in} \wedge \mathcal{E}_B^{out} \quad (\text{Rule 3})$$

$$\frac{A \xrightarrow{(a, p_a)} A', a \notin \mathcal{E}_B}{A \parallel B \xrightarrow{(a, \frac{p_a}{\eta})} A' \parallel B} \quad \frac{B \xrightarrow{(b, p_b)} B', b \notin \mathcal{E}_A}{A \parallel B \xrightarrow{(b, \frac{p_b}{\eta})} A \parallel B'} \quad (\text{Rule 4})$$

Rule 3 determines that synchronisation occurs only when both components are ready to communicate, as input (output) actions wait for a corresponding output (input) action to be ready

for interaction. Rule 4 denotes autonomous moves from each automaton which are interleaved to represent their concurrent execution. A *normalisation factor* η is applied in both cases to preserve the generative semantics of locally controlled actions. We discuss in the next paragraphs the interpretation for η and discuss the differences w.r.t. to the normalisation factors previously described for generative PLTS models, PCIP and PIOA.

Normalisation

Consider the architectural model and the corresponding PCA models for the Client-Server system in Figure 3.16 introduced earlier in this Chapter. The composite PCA model for the Client-Server system depicted in Figure 3.16(b) has been automatically constructed by composing in parallel the PCA representations of the Client and Server components, according to the previous operational semantics for PCA models. Client and Server execute independently their respective internal actions **prepare** and **process**, between the shared actions **request** and **response** through which the Client and Server component synchronise. Although the Server is ready to execute the output action **response** from the composite state $\langle 1, 2' \rangle$, the Server blocks until the Client is ready for synchronisation⁴, following the semantics of parallel composition for Interface Automata. Similarly, in composite state $\langle 2, 1' \rangle$ the Client does not execute the input action **response** until it can synchronise with the corresponding output action from the Server.

To preserve the generative semantics for the composite state $\langle 1, 1' \rangle$, the probabilities of transitions $\langle 1, 1' \rangle \xrightarrow{\text{prepare}} \langle 2, 1' \rangle$ and $\langle 1, 1' \rangle \xrightarrow{\text{process}} \langle 1, 2' \rangle$ in the composite PCA need to be normalised. Note that the interleaving of actions **prepare** and **process** depicts the different execution orders of the concurrent execution of the actions from the composite state $\langle 1, 1' \rangle$ and the composite state $\langle 2, 2' \rangle$ denotes a system state where the two internal actions have both been executed, regardless of their execution order. *In fact, the probability of reaching state $\langle 2, 2' \rangle$ from state $\langle 1, 1' \rangle$ is independent of the normalisation factor used.* As a result, our normalisation factor η does not require any additional information from the system designer as it considers that both components are equally likely to execute their actions before the other. When applied to the Client-Server system, we define η as the sum of the probabilities of all locally controlled actions from the composite state $\langle 1, 1' \rangle$: 2. The corresponding composite PCA is depicted in Figure 3.17.

⁴Asynchronous communication can be modelled using an intermediate buffer component.

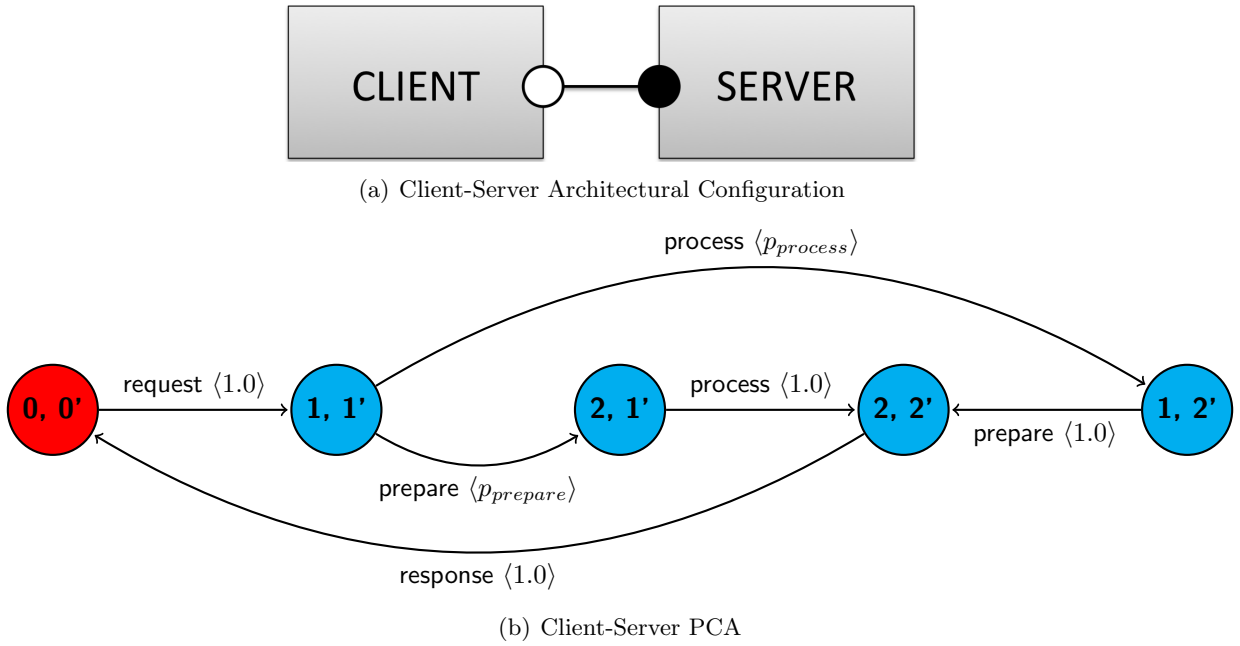


Figure 3.16.: Client-Server System Models

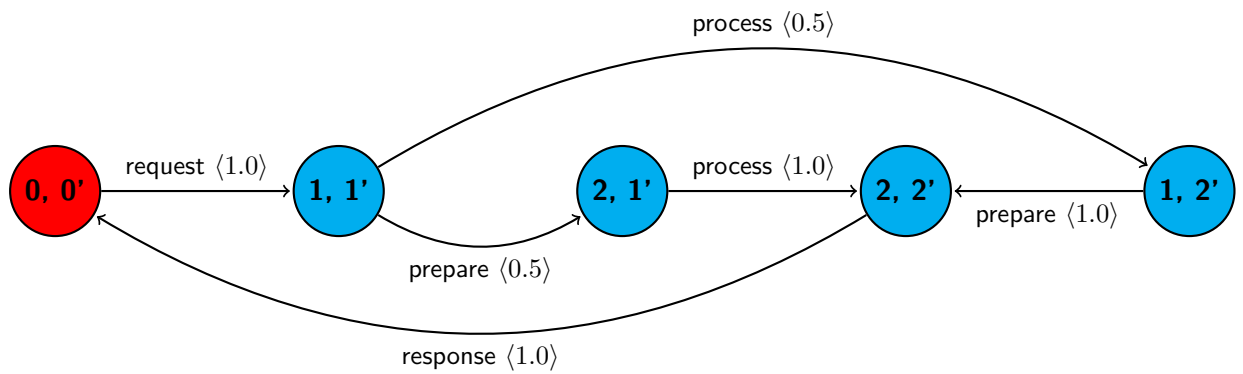


Figure 3.17.: Composite PCA for Client-Server System Using Normalisation Defined by η for PCA Models

We will later distinguish specific cases which require a different definition of the normalisation factor η and formalise their calculation when defining the algorithm that implements the parallel composition operator in Section 3.5.1. In the next paragraphs we illustrate the application of the different normalisation mechanisms described in Section 3.1 and discuss the interpretation of the resulting probabilities in comparison with our normalisation factor η .

Consider the normalisation defined by the parallel composition operator $\parallel_{\sigma,\theta}$ [BBS95] to asynchronous execution of actions. Figure 3.18 depicts the application of the σ factor to the probabilities of transitions $\langle 1, 1' \rangle \xrightarrow{\text{prepare}} \langle 2, 1' \rangle$ and $\langle 1, 1' \rangle \xrightarrow{\text{process}} \langle 1, 2' \rangle$ in the composite PCA in Figure 3.16(b). The normalisation factor σ stipulates in this case the probability that Client finishes the execution of action **prepare** before the Server finishes the execution of action **process**. Given that the Client and Server cannot synchronise on any action from state $\langle 1, 1' \rangle$, the normalisation defined by the parallel composition operator $\parallel_L^\sigma$ is equivalent to $\parallel_{\sigma,\theta}$ in this case. Note that the value of σ does not affect in both cases the probability of reaching state $\langle 2, 2 \rangle$ from state $\langle 1, 1 \rangle$.

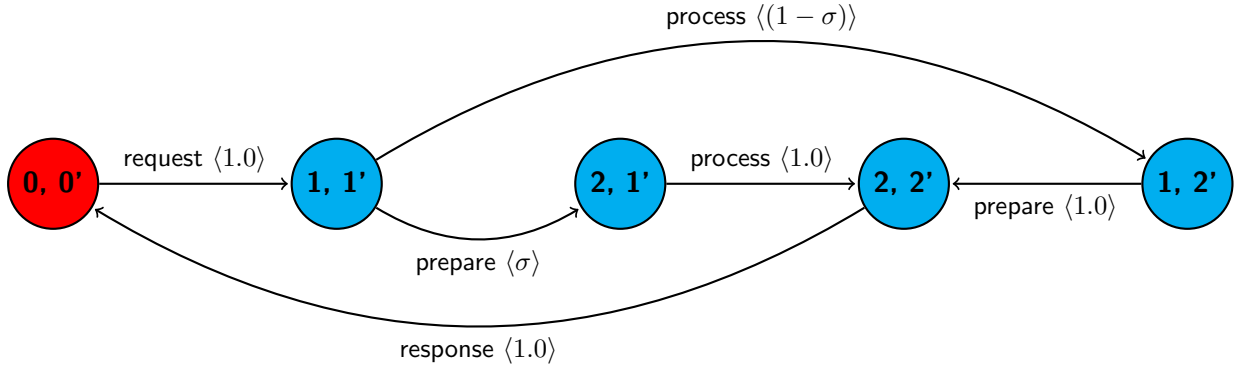


Figure 3.18.: Composite PCA for Client-Server System Using Normalisation Defined by σ [BBS95]

Furthermore, Figure 3.19 depicts the normalisation defined by PIOA based on delay rates of states 1 of Client and $1'$ of Server. Although the delay rate is defined for each state, if the outgoing transitions from each simple state are labelled with the same action, then it can be interpreted as modelling the duration of that action. Consequently, the normalisation defined by $\frac{\delta_{Client}(1)}{\delta_{Client}(1) + \delta_{Server}(1')}$ can be interpreted as expressing the probability that the action **prepare** finishes before the execution of action **process**. However, if one of the components performed more than one internal action from state 1 or $1'$, it is unclear how the normalised transitions would be interpreted as the previous interpretation cannot be directly applied. A similar problem is experienced by PCIP models, where normalisation is based on global delay rates used for all the states of each component, *i.e.* $\frac{\delta_{Client}}{\delta_{Client} + \delta_{Server}}$. Although the authors of PCIP [KGM10] interpreted

the normalisation factor as expressing how frequent a component executes in comparison with others, such interpretation cannot be applied to the probabilities of transitions $\langle 1, 1' \rangle \xrightarrow{\text{prepare}} \langle 2, 1' \rangle$ and $\langle 1, 1' \rangle \xrightarrow{\text{process}} \langle 1, 2' \rangle$ as these denote the concurrent execution of internal action within a session between the Client and the Server. In other words, the Client cannot execute action **prepare** more frequently than the Server executes the action *process* as it can only execute *prepare* after synchronising with the Server through action *request*.

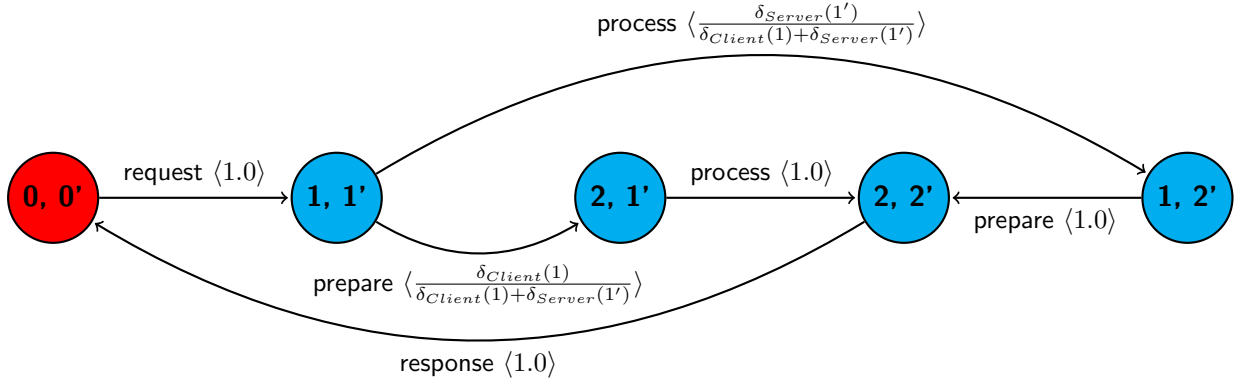


Figure 3.19.: Composite PCA for Client-Server System Using Normalisation Defined by PIOA

In summary, the existing mechanisms to normalise the probability of interleaved transitions labelled with internal or output actions require additional parameters *to achieve the same result as our normalisation factor*. We have performed sensitivity analysis on the different normalisation factors for concurrent execution of internal actions and obtained the same results for properties that involved reachability of failure states. Furthermore, the normalisation used by PIOA and PCIP models also presents interpretation issues regarding the values of delay rates and their relationship with the outgoing transitions from each state. Furthermore, the composite representation of a generic architectural configuration of a system is generally constructed by composing in parallel the models of each component. Given that a configuration may have multiple bindings to a provided interface as well as unbound provided interfaces, we define in the next Sections how the PCA model of each component can be adapted before composition according to the architectural configuration of the system in which it is deployed.

3.3.3. Removing Unused Behaviour

Consider the following alternative Client-Server system as defined in Figure 3.20 and the associate PCA representation for the new Server component in Figure 3.21. This PCA model denotes

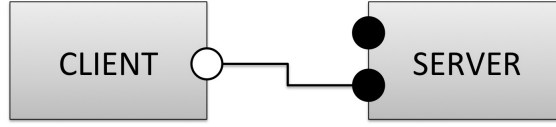


Figure 3.20.: Client-Server System Architecture with Unbound Provided Interface

a generic representation of the new Server component which can be used in any configuration of bindings to its provided interfaces.

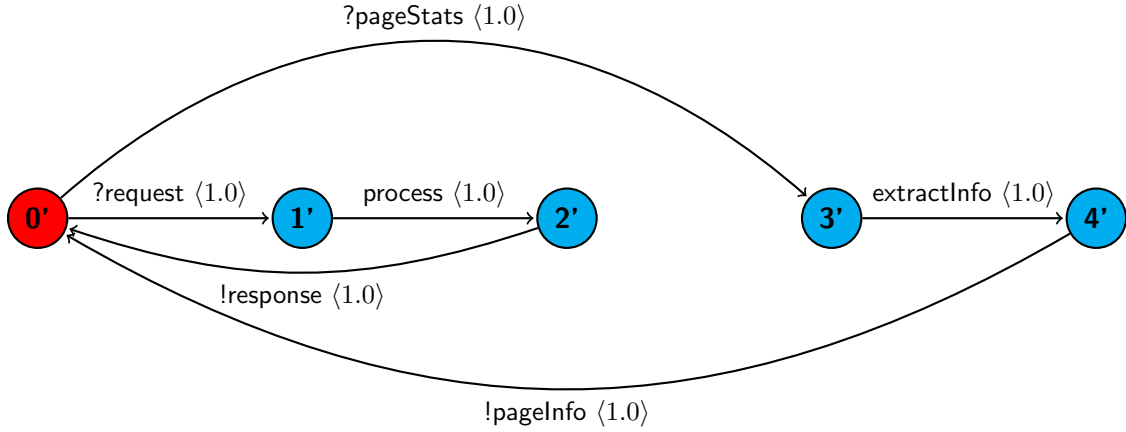


Figure 3.21.: Server PCA with Two Provided Interfaces

However, when this model is composed with the PCA model of the Client component in Figure 3.13(b), then the corresponding composite PCA representation includes unused behaviour associated with the unbound provided interface of the new Server component, as shown in Figure 3.22. As a result, the composite representation includes behaviour that is not going to be used. Therefore, such composite representation cannot be translated to a DTMC model as it includes transitions labelled with input actions which follow reactive semantics.

The interface operator $@$ is used in LTS models to relabel with the silent action τ transitions labelled with actions not in the specified dataset. It is the dual of the hiding operator, which we discuss in the next Section. In contrast, we define the interface operator $@$ to remove unused behaviour associated with unbound provided interfaces. The operational semantics of the operator are defined by Rule 5 and the implementation of this operator is described by Algorithm 3.1.

$$\frac{A \xrightarrow{(a,p_a)} A'}{A}, a \in \mathcal{E}_{unused}^{in} \qquad \frac{A \xrightarrow{(a,p_a)} A'}{A \xrightarrow{(a,p_a)} A'}, a \notin \mathcal{E}_{unused}^{in} \quad (\text{Rule 5})$$

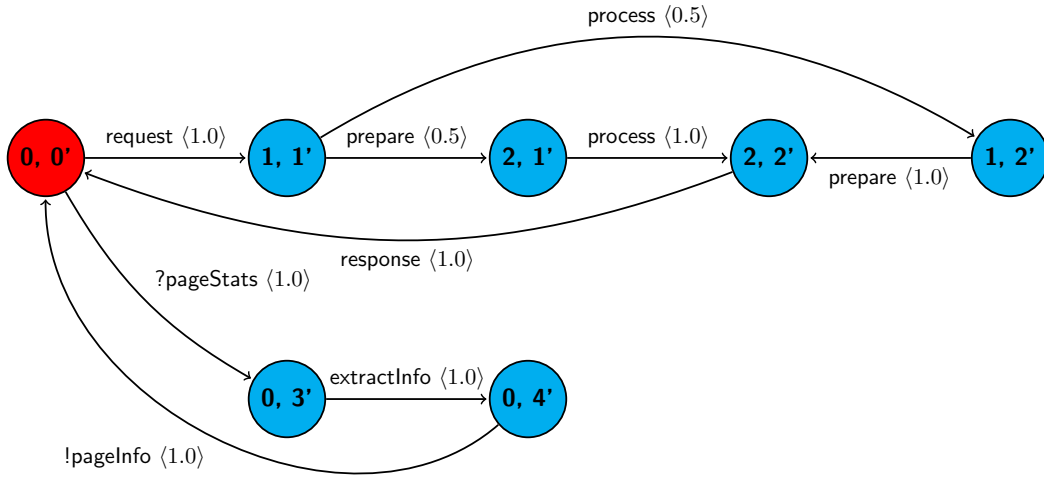


Figure 3.22.: Client-Server Composite PCA with Unused Behaviour

Given an input PCA model $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ and a set of unused input actions $\mathcal{E}_{unused}^{in} \subseteq \mathcal{E}^{in}$, the algorithm performs a depth-first navigation of the PCA model and analyses the outgoing transitions from each state. If a transition is labelled with an input action not in the set $\mathcal{E}_{unused}^{in}$, or any internal, output or failure actions, then it is added to the PCA A' . On the other hand, if a transition is labelled with an input action in $\mathcal{E}_{unused}^{in}$, then such transition is not added to the output PCA model A' and the navigation is not continued from the destination state of that transition. As the execution of an input action is determined by an external automaton, if an input action is not synchronised with a corresponding output action, then any subsequent behaviour is not going to be executed, unless it originates from other paths in the automaton.

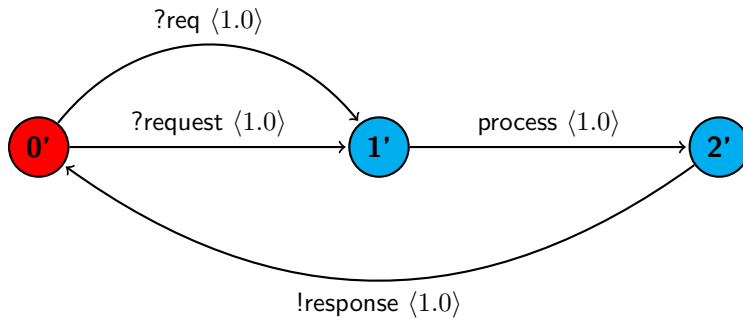


Figure 3.23.: Server PCA with Two Provided Interfaces - Shared Behaviour

Consider an alternative PCA representation in Figure 3.23 for a Server component with two provided interfaces, where the Server includes a second interface associated with input action $?req$ for legacy reasons. In both cases, the PCA model of the new Server can be adjusted

```

input  :  $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$  and  $\mathcal{E}_{unused}^{in}$ 
output:  $A' = \langle \mathcal{S}', q, \mathcal{E}', \Delta', \mu' \rangle$ 

1  boolean[] visited;
2  Queue states;
3  states.push(q);
4  while not states.isEmpty() do
5      currentState ← states.pop();
6      if visited[currentState] then
7          continue;
8      visited[currentState] = true;
9      if  $\Delta(currentState) = \emptyset$  then continue;
10     ;
11     foreach  $(currentState, e, s) \in \Delta(currentState)$  do
12         if  $e \in \mathcal{E}_{unused}^{in}$  then
13             continue;
14         else
15             addTransition((currentState, e, s)) to A';
16         if  $s \neq q \wedge s \neq currentState$  then states.push(s);
17         ;
18     ;

```

Algorithm 3.1: Interface Operator Algorithm

according to the architectural configuration in Figure 3.20 by applying the interface operator ($\text{Server}' @ \{ \text{request} \}$), thus producing a model that is equal to the PCA of the original Server and thus can be used construct a composite representation that reflects the architectural configuration of the system. While in the PCA representation of Figure 3.21, the interface operator discards all paths initiated by transition $0' \xrightarrow{?pageStates} 3'$ as these are never executed by the Server, when applied to the PCA representation in Figure 3.23 the interface operator keeps the behaviour from state $1'$ as the state $1'$ is reached from another path, *i.e.* $0' \xrightarrow{?request} 1'$.

3.3.4. Relabelling

The relabelling operator is used as a means to apply relabelling functions to models in order to change the names of action labels. It is normally used to ensure that models synchronise on the correct action labels when composing them. For example, two components may be bound to compatible provided and required interfaces but there may be a mismatch between the corresponding interface actions. In the general case, the relabelling operator takes as input a relabelling mapping as follows $\{newlabel_1/oldlabel_1, \dots, newlabel_n/oldlabel_n\}$. The operational

semantics of the relabelling operator are defined by Rule 6, where f represents a mapping relation over action labels.

$$\frac{A \xrightarrow{(a, p_a)} A'}{A/f \xrightarrow{f(a, p_a)} A'/f} \quad (\text{Rule 6})$$

When the relation defines a 1-N mapping $\{a\} \times L$, $a \in \mathcal{E}$, each transition $(s, a, s') \in \Delta$ is replaced by $\#(L)$ transitions labelled with the action labels in L . In addition to the re-labelling as used in LTS, the probability associated with those transitions is defined by $\mu(s, a, s')$ in the case of input actions and $\frac{\mu(s, a, s')}{\#(L)}$ for others, where $\mu(s, a, s')$ denotes the probability of reaching state s' from state s through the execution of action a .

Modelling Multiple Bindings

While the previous operators support the specification of basic and composite components with single bindings to each provided interface, the re-labelling operator $/$ can be used to rename transitions labelled with interface actions of a component to support requests from multiple bindings. The components that share the common resource also need to rename their interface actions accordingly so that individual requests from each component can be distinguished.

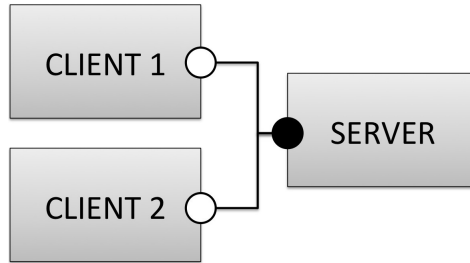


Figure 3.24.: Two Clients-Server Architectural Configuration

When multiple Clients are bound to the Server's provided interface, as shown in Figure 3.24, the Server PCA needs to be extended to handle requests from multiple clients. Figure 3.25 depicts the result of applying the re-labelling operator $/$ to the interface actions **request** and **response** as follows: $\text{Server} / \{ \{c1.\text{request}, c2.\text{request}\}/\text{request}, \{c1.\text{response}, c2.\text{response}\}/\text{response} \}$. The same can be achieved using a *process sharing* operator similar to the one defined for LTS [MK06] which substitutes interface actions of a component by prefixed actions that represent the interaction with multiple bindings: The resulting PCA model constructed as $\{c1, c2\}::\text{Server}$ and is shown in Figure 3.25.

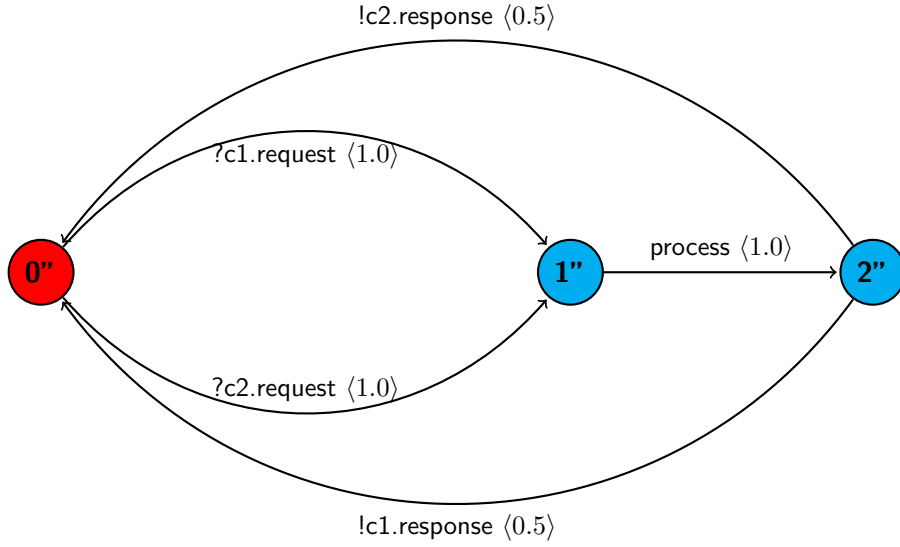


Figure 3.25.: Server PCA with Multiple Bindings

The PCA models of the two clients are derived from the original Client PCA in Figure 3.13(b) by relabelling all its actions in order to fully distinguish their execution. For instance, the PCA model for Client 1 shown in Figure 3.26(a) is constructed as follows: $\text{Client} / \{ \{c1.request\}/request, \{c1.prepare\}/prepare, \{c1.response\}/response \}$. Alternatively, the prefix operator can be used to add a prefix to all the actions of a model. To this end, the model for Client 2 can be constructed by the following P-FSP expression: $c2:\text{Client}$.

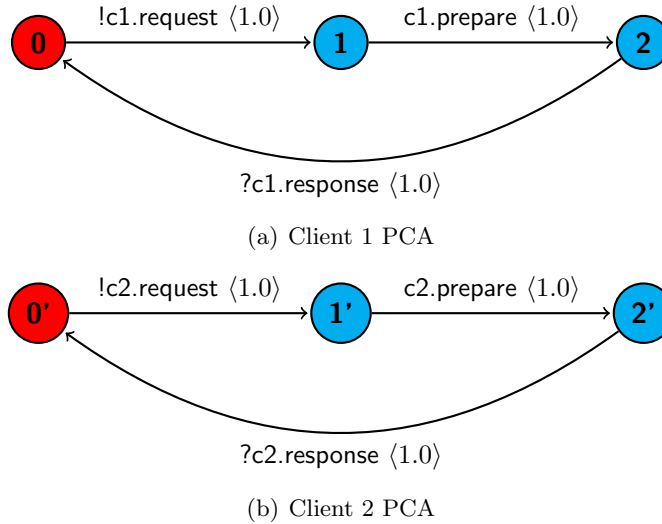


Figure 3.26.: Client 1 and 2 PCA Models

Consider now the composite model constructed using the following P-FSP expression $c1:Client \parallel c2:Client \parallel \{c1, c2\}::Server$ in Figure 3.27. Although the modified Server PCA includes multiple input actions from state $0''$ and multiple output actions from state $2''$, the Server component can only handle one Client at each time. After Client 1 and Server synchronise through action $c1.request$, they execute concurrently their internal actions $c1.prepare$ and $process$, whose probabilities are defined according to the previous rule for normalisation factor η . Note that Client2 needs to wait for the session between Client1 and the Server to finish before it can interact with the Server. Although from the composite state $\langle 2, 0', 2'' \rangle$ the modified Server PCA can execute two output actions $c1.response$ and $c2.response$, the Server can only interact with Client1 as Client2 is waiting to be able to initiate a session with the Server, *i.e.* Client2 can only execute $c2.request$. Consequently, the probability of transition $\langle 2, 0', 2'' \rangle \xrightarrow{c1.response} \langle 0, 0', 0'' \rangle$ needs to be normalised to reflect that the Server can only interact with one Client at a time. However, this scenario represents a different type of normalisation and requires a different definition for the normalisation factor η : the sum of the probabilities of *enabled output actions from a given state*. In the Client-Server example, the second case for normalisation is applied to the transitions labelled with enabled output actions from state $2''$ of Server. *This normalisation corresponds to redistributing the probabilities of blocked transitions labelled with output actions to the remaining transitions with enabled output actions.* It is applied before synchronisation in order to avoid changing: *a)* the original probabilities of internal actions; *b)* the normalisation associated with concurrent execution of internal actions.

Normalisation is also applied at state $\langle 0, 0', 0'' \rangle$ to the requests originating from Client 1 and Client 2. Given that these actions correspond to concurrent requests from the two clients, we have applied the first case of normalisation factor η and the normalised probabilities denote that the two Clients use the Server equally. However, it is important to distinguish this case as normalisation across clients often needs to take into account their differing usage profiles *i.e.* some clients may use the server more frequently than others. Informally, this is analogous to having different normalisation factors for *sessions* originating from different clients and for the interleaving of actions within a session.

Note that three different kinds of normalisation are applied, each with a slightly different meaning. The following cases define also the order in which normalisation is applied.

- in the first case normalisation is applied to a single PCA model to enabled output actions in order for the probabilities of blocked transitions to be distributed to enabled actions;

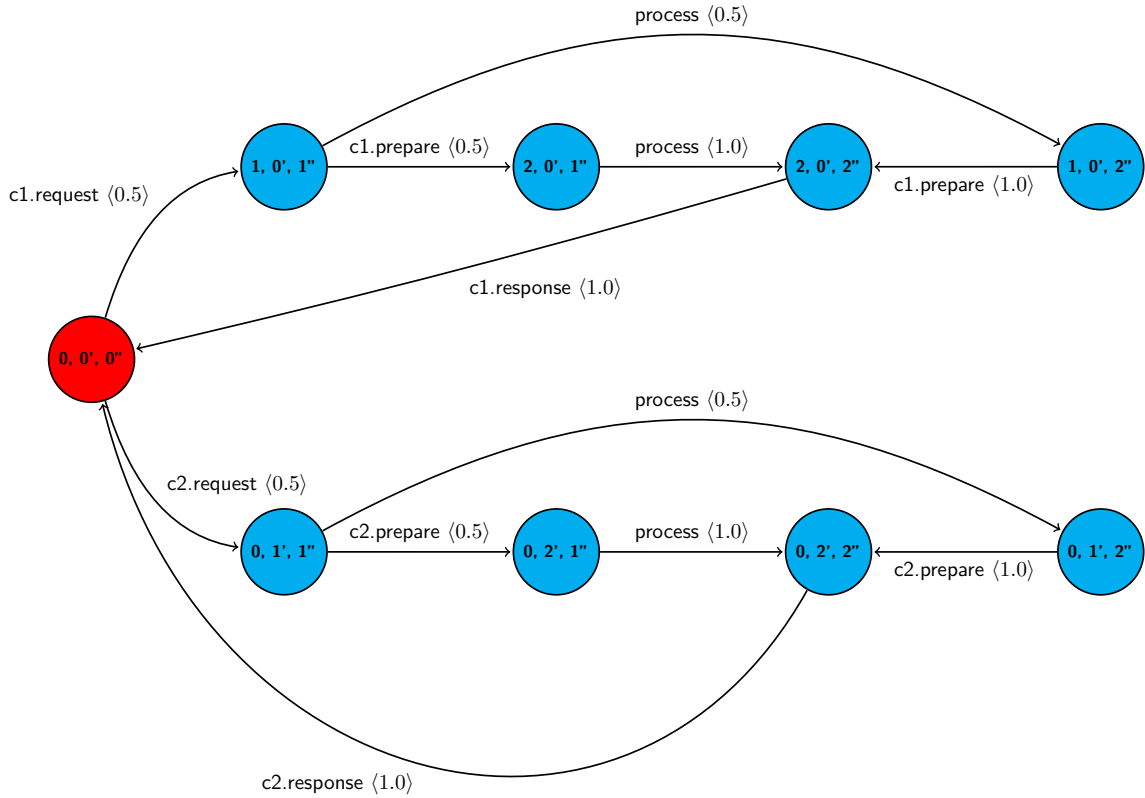


Figure 3.27.: Client-Server Composite PCA with Multiple Bindings

- in the second case normalisation is applied to requests originating from separate clients to model how often they use a shared resource;
- in the third case normalisation is applied to transitions in the composite model labelled with output and internal actions to normalise the probabilities of concurrently executed actions.

The normalisation factors defined for generative PLTS, PCIP and PIOA do not support the presented scenarios as a general rule is applied to all states.

We formalise later in Section 3.5.1 the value of η for the above cases when we describe the algorithm that implements the parallel composition operator. In the next Section we define the constructs for failure modelling in PCA models and the corresponding semantics.

3.4. Reliability Modelling and Analysis

We introduce failure actions to explicitly model failure scenarios, failure propagation and failure handling behaviour. While internal failures represent unexpected executions such as runtime exceptions, transitions labelled with *output failure actions* model externally visible failures, such as communication failures. Both internal and output failures follow generative semantics as they are locally controlled. On the other hand, an *input failure action* $(s, \sim?e, \text{ERROR})$ denotes that a PCA is able to *handle* the failure of the corresponding output action from another component, hence these actions follow reactive semantics as their execution is determined by the PCA that *fails*.

The operational semantics of the parallel composition for failure propagation and failure handling in PCA is intuitively similar to exception handling in Object-Oriented programming languages. An output failure action can be interpreted as an exception being thrown while an input failure action denotes the catch clause in a try-catch block. This allows to express a variety of failure handling behaviours. For example, the failure of an inner component can be handled by an outer component or by another component at the same level. The path initiated by an input failure action can be mapped to the behaviour specified in a *catch* block.

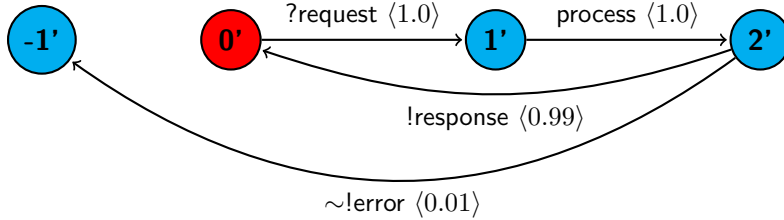


Figure 3.28.: Server PCA with Failures

In Figure 3.28 we show a modified version of the server PCA of our example, where the Server fails with a probability of 1% from state 2' and state -1' denotes an **ERROR** state. When the PCA of the unreliable Server is composed with the Client PCA, the output failure action **error** is *propagated* to the composite PCA and the **ERROR** state in the Server PCA becomes a global **ERROR** state (Figure 3.29). In other words, the failure of a single component, if not handled, leads to the failure of the composite component. Nonetheless, as the transition to the composite failure state is labelled with an external output action, the failure can still be later handled by another component, similar to the way failures are propagated in Object-Oriented languages.

While failure propagation is covered by Rule 4 for non-shared actions, the operational semantics for failure handling is defined in Rule 5.

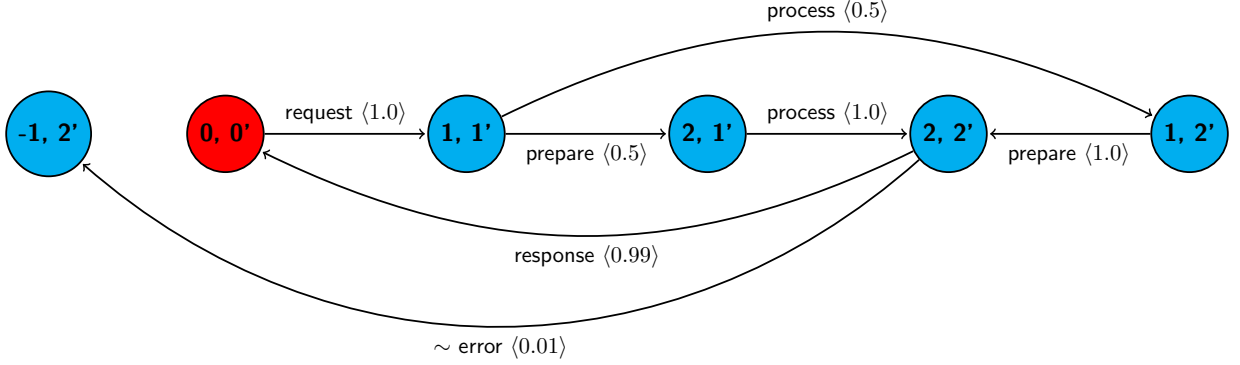


Figure 3.29.: Client-Server Composite PCA with Failures

$$\frac{A \xrightarrow{(\sim!a, p_f)} ERROR, B \xrightarrow{(\sim?a, p_{f_{handling}})} B''}{A || B \xrightarrow{(a, \frac{p_f \cdot p_{f_{handling}}}{\eta})} reset(A) || B''} \quad (\text{Rule 5})$$

Alternatively, the Client PCA can be extended to handle the failure of the *response* action (Figure 3.30) using an input failure action followed by failure handling behaviour (Rule 5). In this case, the composition of the input and output *response* failure actions becomes an internal transition of the composite component that does not lead to the **ERROR** state. Note that the Client can also use an input failure action to prevent a failure of the Server from being handled by other components.

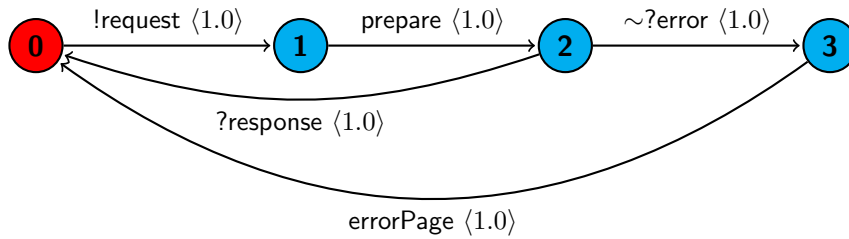


Figure 3.30.: Client PCA with Failure Handling

After the failure action is handled, the Server PCA then *resets* its behaviour to the initial state while the Client PCA continues its execution based on the specified failure handling behaviour. Finally, internal failures are treated in the same way as other internal actions. When two automata are composed, internal failure actions lead the composite automaton to a global **ERROR** state.

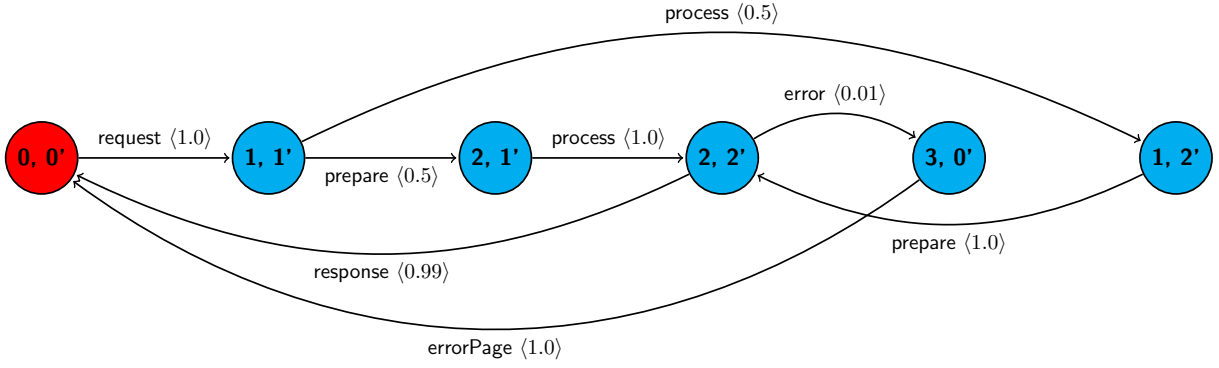


Figure 3.31.: Client-Server Composite PCA with Failure Handling

3.5. Composite Analysis

After describing all the process algebra rules for constructing composite models, we formalise in the next paragraphs how a composite model $A_{\parallel} = A_1 \parallel \dots \parallel A_n$ is constructed by the algorithm that supports the parallel composition operator. We then describe in Section 3.5.3 the *hiding* operator which reduces a PCA model to its interface representation.

3.5.1. Parallel Composition Algorithm

Given n input models $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1, \mu_1 \rangle, \dots, A_n = \langle \mathcal{S}_n, q_n, \mathcal{E}_n, \Delta_n, \mu_n \rangle$, the composite model $A_{\parallel}$ is defined by $\langle \mathcal{S}_{\parallel}, q_{\parallel}, \mathcal{E}_{\parallel}, \Delta_{\parallel}, \mu_{\parallel} \rangle$:

- $\mathcal{S}_{\parallel} \subseteq \mathcal{S}_1 \times \dots \times \mathcal{S}_n$;
- $q_{\parallel} = \langle q_1, \dots, q_n \rangle$;
- $\Delta_{\parallel}$ and $\mu_{\parallel}$ defined below.

The algorithm keeps a queue of states that need to be analysed, which starts with the initial composite state $q_{\parallel} = \langle q_1, \dots, q_n \rangle$. At each step, the algorithm analyses a state $\langle s_1, \dots, s_n \rangle$ and determines which transitions can be executed based on the operational semantics defined earlier in this Chapter. After determining the possible transitions from a given composite state $\langle s_1, \dots, s_n \rangle$, the algorithm calculates their probabilities based on the probabilities defined for individual models and the different cases of the normalisation factor. We describe in the next paragraphs the different cases considered by the algorithm.

As previously mentioned in Section 3.3.2, parallel composition can only be applied to *compatible* PCA models. A PCA $A_i = \langle \mathcal{S}_i, q_i, \mathcal{E}_i, \Delta_i, \mu_i \rangle$ is compatible if $\forall i, j \in [1, n] \wedge i \neq j$, the following conditions are verified:

$$\mathcal{E}_{A_i}^{int} \cap \mathcal{E}_{A_j} = \emptyset, \quad \mathcal{E}_{A_j}^{int} \cap \mathcal{E}_{A_i} = \emptyset,$$

$$\mathcal{E}_{A_i}^{in} \cap \mathcal{E}_{A_j}^{in} = \emptyset, \quad \mathcal{E}_{A_i}^{out} \cap \mathcal{E}_{A_j}^{out} = \emptyset.$$

As a result, synchronisation of input-output pairs can only occur between two models.

Synchronisation Between Input and Output Actions

If two models are able to synchronise from state $\langle s_1, \dots, s_n \rangle$ through a pair of input-output actions, then a transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)$ is added to $\Delta_{\parallel}$, an internal action a is added to $\mathcal{E}_{\parallel}^{int}$ and the composite state $\langle s'_1, \dots, s'_n \rangle$ is added to the queue for further analysis. The two models A_i and A_j can synchronise if:

- $(s_i, a, s'_i) \in \Delta_i$ and $(s_j, a, s'_j) \in \Delta_j$;
- $(a \in \mathcal{E}_i^{in} \wedge a \in \mathcal{E}_j^{out})$ or $(a \in \mathcal{E}_j^{in} \wedge a \in \mathcal{E}_i^{out})$.

Given that only two models A_i and A_j can make a synchronous move, the composite state $\langle s'_1, \dots, s'_n \rangle$, is defined as for all other models $k \in [1, n] \wedge k \neq i \wedge k \neq j$ as $s'_k = s_k$. The same construction applies to all of the following rules.

Synchronisation Between Input Failure and Output Failure Actions

Failure handling is represented by the synchronisation between an input-failure action and the corresponding output-failure action. In the case that a model A_i handles a failure from a model A_j , a transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)$ where $s'_j = q_j$ is added to $\Delta_{\parallel}$ as model A_j has been *reset*, an internal action a is added to $\mathcal{E}_{\parallel}^{int}$ as the failure has been handled and the composite state $\langle s'_1, \dots, s'_n \rangle$ is added to the queue for further analysis. Failure handling occurs if:

- $(s_i, a, s'_i) \in \Delta_i$ and $(s_j, a, s'_j) \in \Delta_j$;
- $(a \in \mathcal{E}_i^{fail-in} \wedge a \in \mathcal{E}_j^{fail-out})$.

Non-shared Actions

We now consider transitions labelled with non-shared actions. A transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)$ is added to $\Delta_{\parallel}$, where $s'_j = s_j \ \forall j \in [1, n] \wedge j \neq i$ as only model A_i progresses. The composite state $\langle s'_1, \dots, s'_n \rangle$ is then added to the queue for further analysis. If a is a non-shared input action, *i.e.* $a \in \mathcal{E}_i^{in} \wedge a \notin \mathcal{E}_j^{out}, \forall j \in [1, n] \wedge j \neq i$, then a is added to $\mathcal{E}_{\parallel}^{in}$. Similarly, if a is a non-shared output action, *i.e.* $a \in \mathcal{E}_i^{out} \wedge a \notin \mathcal{E}_j^{in}, \forall j \in [1, n] \wedge j \neq i$, then a is added to $\mathcal{E}_{\parallel}^{out}$. In the case of internal actions, a transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)$ is added to $\Delta_{\parallel}$ and the action label a is added to $\mathcal{E}_{\parallel}^{int}$.

We define in the next paragraphs how the probabilities of composite transitions are defined, including the probability of transitions from a state with blocked transitions.

Probability of Composite Transitions - $\mu_{\parallel}$

We now define the probabilities of transitions in the composite model $\Delta_{\parallel}$. We start by determining the probability of transitions labelled with input actions that are not synchronised, as these are the only that do not require further normalisation.

From a state $\langle s_1, \dots, s_n \rangle$, the probability of a transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle) \in \Delta_{\parallel}$ is defined by $\mu_i(s_i, a, s'_i)$ when the input action is *executed* by model A_i . Similarly, from a state $\langle s_1, \dots, s_n \rangle$, the probability of a transition $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle) \in \Delta_{\parallel}$ is defined by $\mu_i(s_i, a, s'_i)$ when the internal action is *executed* by model A_i . However, $\mu_i(s_i, a, s'_i)$ may not be the final probability in the composite model as it may have to be normalised.

Furthermore, transitions labelled with shared output actions cannot execute when the corresponding input action is not ready for synchronisation. We define $enabled(s_i, s_1, \dots, s_n)$ as the set of output actions from state s_i that are not blocked by each model A_j from state s_j , where $j \in [1, n] \wedge j \neq i$:

$$enabled(s_i, s_1, \dots, s_n) = \{a \in \Delta_i^e(s_i) \wedge a \in \mathcal{E}_j^{out} : \exists s_j \in (s_1, \dots, s_n) \wedge (a \in \Delta_j^e(s_j) \wedge a \in \mathcal{E}_j^{in}) \vee (a \notin \mathcal{E}_j)\}.$$

Consequently, output actions in the *enabled* set may need to be normalised before synchronising with input actions. From a composite state $\langle s_1, \dots, s_n \rangle$, the probability of output actions from state s_i of A_i are normalised as follows:

$$\forall (s_i, a, s'_i), a \in \mathcal{E}_i^{out}, \mu'_i(s_i, a, s'_i) = \frac{\mu_i(s_i, a, s'_i)}{\eta^{enabled}(s_i, s_1, \dots, s_n)},$$

where $\eta^{enabled}(s_i, s_1, \dots, s_n)$ is defined as

$$\eta^{enabled}(s_i, s_1, \dots, s_n) = \frac{\mu_i^{enabled}(s_i, s_1, \dots, s_n)}{\sum_{(s_i, a^{out}, s'_i) \in \Delta(s_i)} \mu_i(s_i, a^{out}, s'_i)},$$

and $\mu_i^{enabled}(s_i, s_1, \dots, s_n)$ is defined as

$$\mu_i^{enabled}(s_i, s_1, \dots, s_n) = \sum_{\substack{(s_i, a, s'_i) \in \Delta_i \\ a \in enabled(s_i, s_1, \dots, s_n)}} \mu_i(s_i, a, s'_i)$$

This normalisation effectively corresponds to distributing the probability of blocked output actions to enabled output actions.

Furthermore, the probability of synchronised transitions $(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)$ is firstly defined as:

$$\mu'_{\parallel}(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle) = \left(w_i \cdot \mu'_i(s_i, a, s'_i) \right) \cdot \mu_j(s_j, a, s'_j), \quad \text{if } a \in \mathcal{E}_j^{in} \wedge a \in \mathcal{E}_i^{out},$$

where w_i denotes the weight associated with normalisation of actions in multiple bindings scenarios. Otherwise w_i is equal to 1 for composite models that do not involve multiple bindings to provided interfaces. After the previous two normalisation cases are applied, the probability of locally controlled actions that can be executed from a given composite state $\langle s_1, \dots, s_n \rangle$ is normalised to preserve the generative semantics associated with these transitions as follows:

$$\mu_{\parallel}(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle) = \frac{\mu'_{\parallel}(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)}{\sum_{(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle) \in \Delta_{\parallel}(\langle s_1, \dots, s_n \rangle)} \mu'_{\parallel}(\langle s_1, \dots, s_n \rangle, a, \langle s'_1, \dots, s'_n \rangle)}.$$

Note that the existing mechanisms in the literature for normalising the probability of transitions labelled with internal or output actions cannot cope with all the above cases as they are based on a single rule which focuses on normalising independent choices from different models, *e.g.* interleaved transitions.

3.5.2. Reliability Analysis

A composite PCA that does not have input actions can be translated to a DTMC model which we annotate with state variables **finish** and **fail** to represent successful executions and the occurrence of failures. The probability of the system successfully executing without failures for a single request can be automatically analysed using the translated DTMC model based on the

following PCTL formula: $1 - \mathcal{P}_{=?}[s = \text{fail} \cup \neg(s = \text{finish})]$, the complement probability of the system failing for a single request. Note that this analysis can be generalised to verify how likely the system is to fail after N requests and additional variables can be added to the model to enable sensitivity analysis on the impact of changes on a) the probability of failure of a component; b) clients execution profile and c) bindings configuration.

3.5.3. Hiding

The composite model for a complex system often large as it contains many internal states and transitions. We define the hiding operator $\backslash\{a_1, \dots, a_n\}$ to remove, when possible, the transitions in a PCA A labelled with the internal action $\{a_1, \dots, a_n\}$ and propagates their probability to the remaining transitions, while maintaining the probabilistic reachability properties of the original process. If all the transitions labelled with internal actions are removed from the original model, then the hiding operator reduces a PCA to its interface behaviour representation, *i.e.* with only input and output actions that denote interactions via provided and required interfaces. Removing transitions labelled with internal actions is more complex than removing unused behaviour as it requires propagating the probability of the deleted transitions whilst preserving the reachability properties of the original model. We devote the the entirety of the next Chapter to the algorithm that implements the hiding operator to help reduce the state space, thereby mitigating the state-explosion problem associated with composite representations.

Note that we apply a different semantics to the hiding operator that is generally defined for other process algebras where it is used to replace action labels with a special action label τ . In those process algebras, actions that are relabelled with τ can no longer be used to synchronise with other models. This is not necessary for PCA models as the parallel composition operator rules prevent synchronisation involving internal actions.

3.5.4. Profiling

The probabilities associated with internal and output actions of a PCA representation can be obtained by profiling the system and counting the number of times each action was executed over a certain period of time. The initial probabilities can be extracted by simulating or running the system for an initial bootstrapping period. As the execution profile of each component can change at run-time, the initial system representation can become inaccurate. Whilst it is possible to count the number of actions executed at run-time, this information still needs to be integrated

with the initial profile. This has been addressed by Epifani *et. al* [EGMT09] in the case of a single DTMC representation for the whole system. A new transition matrix P' (posterior) is calculated from the previous transition matrix P (prior) and the recent information extracted from execution traces (likelihood of each transition). This method can be analogously applied to update the probabilities associated with the internal and output actions of a PCA representation. In contrast with DTMC models, as profiling for PCA models is performed independently by each component, *when a component is replaced the execution profile of the remaining components can be re-used to construct the composite model of the new configuration*. Moreover, as PCA representations take into account the component's internal behaviour, changes in the execution profile of a single component can be detected sooner, thus improving the accuracy of the overall system representation.

3.6. Tool Support - LTSA-PCA

In this Section we briefly describe our extension the LTSA model checker [MK06] to support the specification, construction and analysis of PCA models. The full description of LTSA-PCA is given in Appendix B.

In the same way that LTS models are incrementally specified using Finite State Processes (FSP), we have defined the extension Probabilistic Finite State Processes (P-FSP) to support the specification of PCA models. All the operators described in this Chapter have been implemented in LTSA-PCA according to the operational semantics defined in the previous sections. For example, the prefix and choice operators of FSP have been extended to consider the type of each action as well as its probability. The behaviour of each component is specified using a P-FSP expression which is then compiled to a PCA model. The parallel composition operator is then used to automatically construct the composite PCA model of a given system. Each model can be adjusted prior to composition using the interface and relabelling operators in order to reflect the architectural configuration. Automated reliability analysis is performed by translating a closed composite PCA to a DTMC model and then analysing its reliability properties in PRISM model checker [KNP]. Finally, behavioural analysis is supported by existing tools in LTSA.

3.7. Summary and Related Work

Modelling the probabilistic behaviour of distributed systems is a challenging problem. It requires finding a reasonable interpretation of probabilities for the representation for each component, which is then preserved when composing models to construct the representation of a system configuration. Hybrid models that associate reactive semantics with input actions and generative semantics with internal and output actions provide an important contribution by enabling the representation of behaviour that a component controls and actions whose execution is chosen by others. However, the composition semantics of existing hybrid models hinders their applicability to model the behaviour of composite components.

In addition, composable modelling of failure scenarios is also very limited. Although failure actions can be modelled using internal actions that lead to an absorbing *failure* state, this approach does not support the representation of failure handling behaviour. Existing approaches also do not consider in detail the implications of the architectural model when constructing the composite model for a system configuration, which results in the following limitations:

- if the configuration comprises unbound provided interfaces, then the composite model cannot be used for exact probabilistic analysis as it includes unresolved non-deterministic choices associated with input actions that are not used by other components;
- there is no support for configurations comprising multiple components bound to a single provided interface as only single bindings to provided interfaces are supported. Even if best- and worst-case analysis is used, as in the case of MDPs, the analysis would not be correct as it would be based on a model includes behaviour that is never used.

Probabilistic Component Automata overcome the limitations of existing models by combining and extending elements from existing work in a coherent model that corresponds closely to architectural component models based on Architecture Description Languages, such as Darwin [MDEK95]. By establishing a close link with architectural models, the individual model of each component can be automatically adjusted according to a specific configuration of bindings. In addition, explicit modelling for failure actions is supported by PCA representations and the associated semantics follow closely programming models in object-oriented languages and distributed component models. In addition to providing an intuitive semantics for modelling the probabilistic behaviour of basic components, we defined a suitable semantics for construct-

ing composite probabilistic behaviour representations. Therefore, the specification burden for complex systems can be reduced by modelling each component individually.

Furthermore, the use of fine-grained representations that take into account the internal behaviour of components also improves the responsiveness of reliability analysis. Since the usage profile of provided interfaces of an inner component is determined by outer component(s), the reliability provided by the inner component can only be computed when the probabilities associated with the equivalent output actions in the automaton of an outer component are propagated to the input actions in the composite automaton. If the probabilities associated with these output actions change, the new reliability values can be readily computed thus facilitating the detection of violations of reliability requirements. In contrast, if internal behaviour is not taken into account and the overall system representation is not automatically constructed using the representation of its parts, new reliability values can only be measured after enough data that reflects the new behaviour of the composite component has been collected by profiling tools.

The compatibility of two components with respect to their interface behaviour (see Section 2.2.2) can be verified using the model checking tools available for LTS representations [MK06] extracting the underlying IA model from the composite PCA model. Although we use the same techniques as IA for analysing the compatibility between two components, we modify the original PCA models before composition in order to reflect the architectural configuration of the system.

Although the semantics of the underlying formalisms on which they are based is different, there are close similarities between PCA models and performance process algebras such as PEPA [Hil96]. Informally, in PEPA, a composite performance model is constructed from the representations of its parts, *rates* are used instead of probabilities for locally controlled actions and *passive rates* are assigned to passive actions, which are analogous to input actions. Additionally, in PEPA only a model without any passive rates can be used for performance analysis, as a composite model without passive actions can be translated to a CTMC representation. While PEPA can be informally seen as a compositional formalism to build a composite CTMC model of a system for analysis of its performance properties, PCA can be seen as a compositional formalism to construct the composite DTMC model of a system for analysis of its reliability properties.

Notwithstanding that composability facilitates incremental elaboration and enables the definition of more fine grained representations, due to the interleaving of internal actions, the

composite PCA for a large system may suffer from state-explosion. We discuss how we can contribute to address this problem in the next Chapter.

4. Towards Scalable Compositional Reliability Analysis

Composability is a fundamental aspect for automating analysis of alternative architectural configurations based on composite models that are automatically constructed from the representations of individual components. However, due to the interleaving of actions that are concurrently executed in different components, composite models may face state-explosion which hinders the scalability of the analysis. Although fine-grained detail is needed for the specification and profiling of the probabilistic behaviour of each component, each PCA model can be simplified to its interface behaviour before composition to reduce the number of interleaved transitions, thereby mitigating the state-explosion problem. Such reduction process requires a minimisation technique to remove internal behaviour whilst preserving the reachability properties of the original model.

Compositional Reachability Analysis (CRA) [CK96] has been proposed to help mitigate the state-explosion associated with the composition of LTS and thereby potentially improving the scalability of their composition [GS91]. Consider two LTS $Q = \langle \mathcal{S}_Q, q_Q, \mathcal{E}_Q, \Delta_Q \rangle$ and $P = \langle \mathcal{S}_P, q_P, \mathcal{E}_P, \Delta_P \rangle$ representing the behaviour of two components. In the composite behaviour $Q||P$, Q imposes behaviour contextual constraints on P [CK96]. Such constraints are captured by an interface process I such that the properties which hold for $Q||I$ also hold for $Q||P$. This requires that I is behaviourally equivalent to $P \upharpoonright \alpha_Q$, *i.e.* a process that is constructed by restricting P to the actions in Q . The main steps for building the interface process I of P constrained by Q are as follows:

- for every transition $(s, e, s') \in \Delta_P$, if $e \notin \mathcal{E}_Q$ delete transition (s, e, s') ;
- merge states s and s' into a single state.

However, this CRA method [CK96] cannot be directly applied to PCA representations, or probabilistic models in general, as it does not take into account the probability of deleted transitions. How can the probability of deleted transitions be propagated to remaining transitions?

In this Chapter we propose an algorithm that extends the principles of CRA to probabilistic generative systems and discuss its application to reduce PCA models to their interface behaviour. We describe the steps taken by the algorithm to delete transitions and evaluate its computational complexity. Using a series of small examples we empirically show that the reduced models preserve the reachability properties of the original representations. We further use two example WebServer systems to illustrate the size reduction obtained by the algorithm. The formal verification of the correctness of the algorithm is discussed in Section 4.3 and its full specification is given in Appendix A.

4.1. Reduction Algorithm

Before presenting the reduction algorithm for PCA models the following definitions are required based on a generic PCA model $\langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$:

- $\Delta(s) = \{(s, e, s') \in \Delta \mid s' \in \mathcal{S}, e \in \mathcal{E}\}$: the successor transitions of a state $s \in \mathcal{S}$;
- $\rho(s) = \{(s', e, s) \in \Delta \mid s' \in \mathcal{S}, e \in \mathcal{E}\}$: the predecessor transitions of a state $s \in \mathcal{S}$;
- $\Delta_s(s) = \{s' \mid (s, e, s') \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the states of successor transitions of a state $s \in \mathcal{S}$;
- $\rho_s(s) = \{s' \mid (s', e, s) \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: states of predecessor transitions of a state $s \in \mathcal{S}$;
- $\Delta_e(s) = \{e \mid (s, e, s') \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the actions of successor transitions of a state $s \in \mathcal{S}$;
- $\rho_e(s) = \{e \mid (s', e, s) \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the actions of predecessor transitions of a state $s \in \mathcal{S}$;

input : $P = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ and $\mathcal{E}_{remove}^{int} \subseteq \mathcal{E}_P^{int}$

output: $I_P = \langle \mathcal{S}_I, q_I, \mathcal{E}_I, \Delta_I, \mu_I \rangle$

1 $[P', \text{markedStates}, \text{cyclicPaths}] = \text{firstPhase}(P, \mathcal{E}_{remove}^{int})$

2 $I_P = \text{secondPhase}(P', \text{markedStates}, \text{cyclicPaths}, \mathcal{E}_{remove}^{int})$

Algorithm 4.1: PCA Reduction Algorithm

The reduction algorithm (Algorithm 4.1) receives as input a PCA model $P = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ and a sub-set of its internal actions $\mathcal{E}_{remove}^{int}$. Note that it may be desirable to keep transitions labelled with certain input actions for the analysis of properties that require those actions. The algorithm analyses the transitions of each state and checks if transitions labelled with internal actions in the set $\mathcal{E}_{remove}^{int}$ can be deleted based on whether their probabilities can be propagated *forwards* to subsequent transitions or *backwards* to prior transitions. In the next paragraphs we discuss the different situations analysed by the algorithm before describing how an input model is reduced.

Firstly, transitions labelled with input or output actions must be kept by the algorithm as these are necessary for synchronisation with other components. Input actions that are not used for synchronisation can be removed using the interface operator introduced in the previous Chapter, but no propagation of probabilities is needed in such case. Additionally, transitions labelled with internal actions which are not included in the input action set $\mathcal{E}_{remove}^{int}$ and self-loop transitions that start and end at the same state must also be kept. We avoid removing these transitions as their probabilities cannot be simply propagated. Alternatively, we could calculate the steady-state probabilities and then propagate them but such calculation would significantly increase the complexity of the reduction algorithm as it requires information regarding the entire model. In contrast, the deletion of non-cyclic transitions only requires local information. Formally, the algorithm keeps a transition $(s, a, s') \in \Delta$ if one of the following conditions holds:

$$a \notin \mathcal{E}_{remove}^{int} \tag{4.1}$$

$$s = s' \text{ (self-loop transition)} \tag{4.2}$$

For single non-cyclic transitions labelled with an internal action in $\mathcal{E}_{remove}^{int}$, the algorithm first tries to delete a single transition labelled with an internal action in $\mathcal{E}_{remove}^{int}$ and its probability is **propagated forward** if its destination state does not have other incoming transitions. In the example model of Figure 4.1, the transition $s \xrightarrow{a} s'$ can be considered for deletion as the state s' does not have other incoming transitions.

On the other hand, if state s' has other incoming transitions, these can still be deleted provided that all those transitions are labelled with internal actions in $\mathcal{E}_{remove}^{int}$ and they all come from the same source state s . This case is illustrated by Figure 4.2 where transitions $s \xrightarrow{a} s'$, $s \xrightarrow{a'} s'$ and $s \xrightarrow{a''} s'$ verify the previous conditions and can therefore be considered by the algorithm

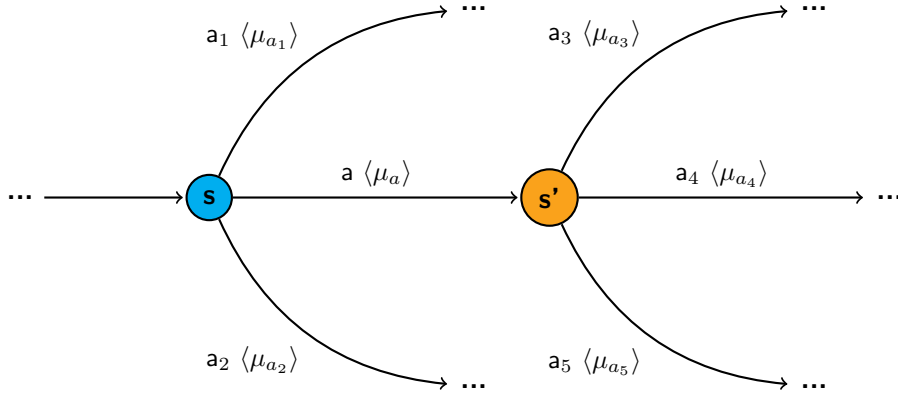


Figure 4.1.: Forward Propagation of a Single Transition

for deletion and their aggregated probability propagated forward to the outgoing transitions of state s' .

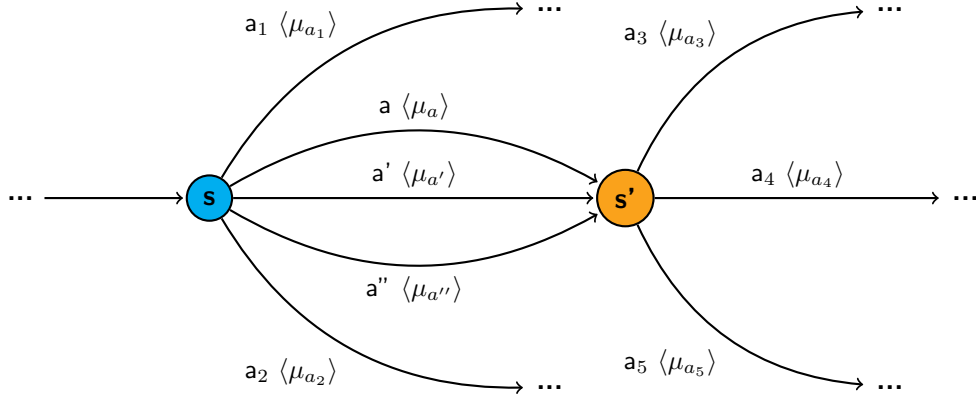


Figure 4.2.: Forward Propagation of Multiple Transitions

Transitions (s, a_i, s') are therefore *propagated forward* only iff **all** the following conditions hold:

$$\#(\rho_e(s')) \subseteq \mathcal{E}_{remove}^{int} \text{ (Incoming transitions are labelled with actions in the set } \mathcal{E}_{remove}^{int} \text{)} \quad (4.3)$$

$$\#(\rho_s(s')) = 1 \text{ (Incoming transitions to state } s' \text{ come from the same source state)} \quad (4.4)$$

$$\Delta(s) \neq \emptyset \text{ (Existence of transitions for forward propagation)} \quad (4.5)$$

$$\Delta_e(s') \cap \mathcal{E}^{in} = \emptyset \text{ (Invalid propagation to input actions)} \quad (4.6)$$

$$s' \neq q \text{ (Invalid propagation to outgoing transitions of initial state } q \text{)} \quad (4.7)$$

$$\Delta_e(s') \cap \mathcal{E}^{out} = \emptyset \text{ (Invalid propagation to output actions)} \quad (4.8)$$

The first condition 4.3 prevents the removal of transitions labelled with actions not in the deletion set $\mathcal{E}_{remove}^{int}$. Conditions 4.4 and 4.5 ensure that the probability of incoming transitions to state s' can be propagated forward and that there are outgoing transitions from state s' to which the aggregated probability can be propagated. Condition 4.6 prevents probabilities from being propagated to transitions labelled with input actions to avoid violation of their reactive semantics. Similarly, condition 4.7 prevents forward propagation to outgoing transitions of the initial state *as the transitions to be deleted cannot be executed before the outgoing transitions of the initial state are executed*. Additionally, the transitions are not deleted if the destination state s' has outgoing transitions labelled with output actions, as defined by condition 4.8. *Although this situation does not affect the correctness of individual models, it changes the probabilities when composing with other models*. We discuss this in more detail in Section 4.2.2 and in the Appendix A.

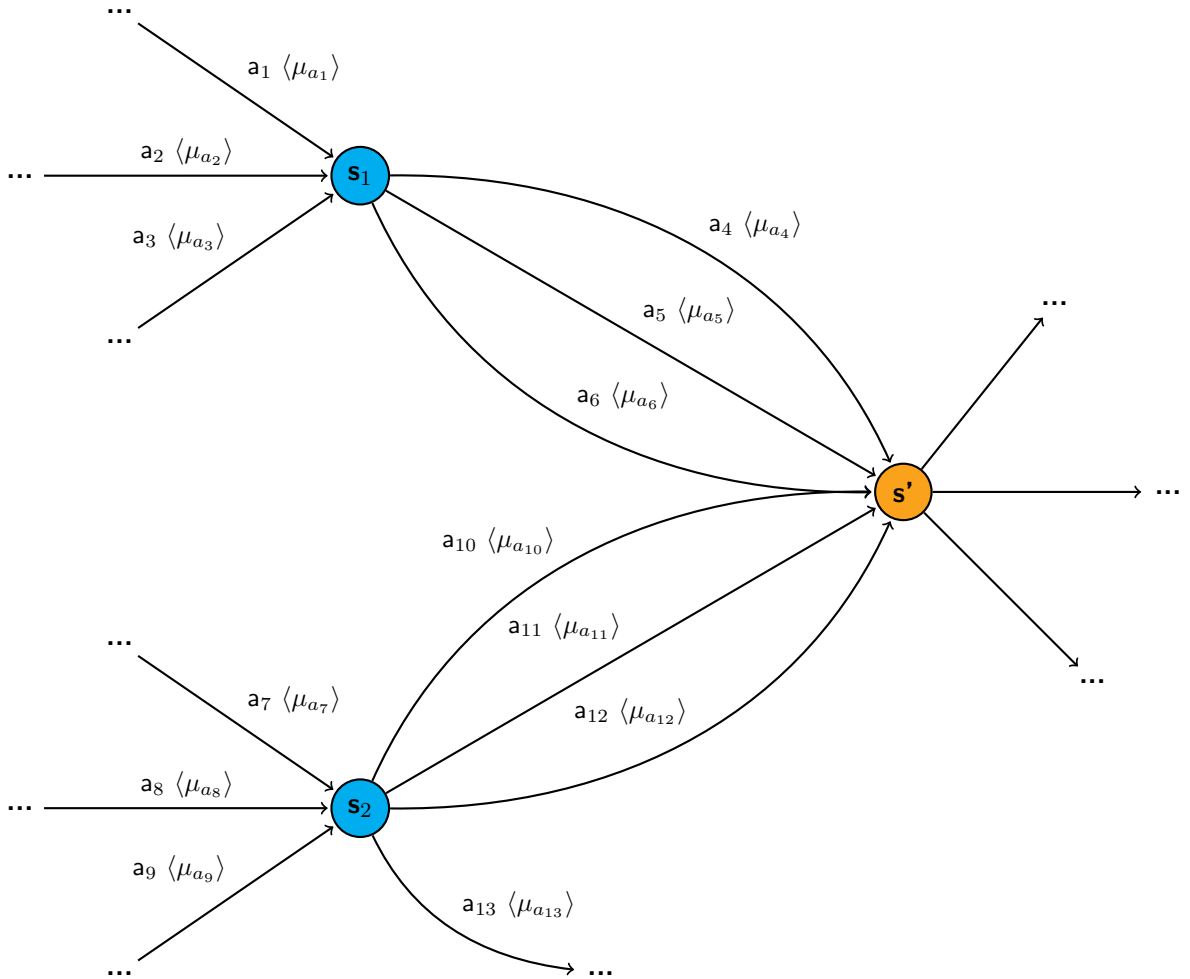


Figure 4.3.: Backward Propagation of Multiple Transitions

If one of the above conditions is not verified, the algorithm tries to *propagate the transitions backwards*. It first groups incoming transitions to state s' based on their source state and then analyses them separately. Each group of transitions (s, a_i, s') with the same source state s is *deleted* and their aggregated probability *propagated backwards* iff all the following conditions are satisfied:

$$\#(\Delta_e(s)) \subseteq \mathcal{E}_{remove}^{int} \text{ (Transitions from state } s \text{ are labelled with actions in } \mathcal{E}_{remove}^{int} \text{)} \quad (4.9)$$

$$\#(\Delta_s(s)) = \{s'\} \text{ (All transitions from state } s \text{ have } s' \text{ as the destination state)} \quad (4.10)$$

$$s \neq q \text{ (Invalid propagation to incoming transitions to the initial state } q \text{)} \quad (4.11)$$

For each group of transitions that originate from the same source state, conditions 4.9 and 4.10 ensure that the algorithm only removes them if they are all labelled with internal actions in the set $\mathcal{E}_{remove}^{int}$ and the source state does not have other outgoing transitions. This is the case of state s_1 in Figure 4.3. Note the conditions 4.6 and 4.8 required for forward propagation do not need to be verified in the case of backward propagation as condition 4.10 implies that the aggregated probability of the transitions to be deleted is 1. Therefore, the probability of transitions labelled with input actions is not changed. Moreover, condition 4.11 *prevents backward propagation to incoming transitions of the initial state as the transitions to the incoming transitions to the initial state are only executed after the deleted transitions are executed*.

Similarly to the original CRA method for LTS models, when transitions (s, a_i, s') are deleted the states s and s' are merged. In other words, the outgoing transitions of state s' are updated to originate from state s when the probability deleted transitions is propagated forward, and the incoming transitions to state s are updated to have as destination state s' when the probability of deleted transitions is propagated backwards.

4.1.1. Algorithm Description

The process of reducing a PCA model P is divided in two phases, where the transitions of each state of model P are analysed to determine which transitions can be removed. During the first phase (Algorithm 4.2), the algorithm visits all the states and transitions of the input model P using a depth-first search to delete, when possible, single transitions labelled with internal

actions. The following information is also collected during the first phase which is later used in the second phase:

- states with multiple incoming transitions;
- the number of cyclic paths that start (and by definition end) at a given state, which also includes self-loop transitions.

The navigation process described in Algorithm 4.2 starts from the initial state by adding it to the queue of states to be analysed (line 3 - Algorithm 4.2) and the navigation continues as long as there are states in the queue to be analysed (line 4 - Algorithm 4.2). After removing the next state to be analysed from the queue (*currentState* in line 5 - Algorithm 4.2), the first step consists in specifying that the state *currentState* has been visited (line 7 - Algorithm 4.2) to ensure the outgoing transitions of each state are traversed only once. The algorithm then analyses all its outgoing transitions $\Delta(\textit{currentState})$ (line 8 - Algorithm 4.2) and decides whether each transition is kept in the intermediate reduced model P' according to the previously defined conditions for forward propagation (lines 11-18 - Algorithm 4.2). Finally, when a transition labelled with an internal action in $\mathcal{E}_{\textit{remove}}^{\textit{int}}$ cannot be deleted in the first phase of the algorithm (lines 19-21, Algorithm 4.2), it is added to the intermediate reduced model and its destination state s is marked for later analysis. After analysing each transition, if its destination state has already been visited, the algorithm needs to determine whether this refers to multiple paths merging at the same state or if it is a cyclic path (lines 21-26 - Algorithm 4.2). This information is paramount for the second phase of the algorithm to ensure the navigation progresses in the presence of cyclic paths.

The second phase takes as input the intermediate reduced model P' produced by the first phase along with the number of cyclic paths for each state and the states marked for analysis. While in the first phase the input PCA model was traversed using a depth-first algorithm, the second-phase uses a breadth-first navigation to guarantee that a state is only considered for analysis after all its incoming transitions have been traversed. These transitions are only traversed when the incoming transitions to their source state have been examined for deletion. *The recursive application of this condition ensures that the incoming transitions of a state are only analysed when all the possible reductions up to the source state of each of those transitions have been applied.*

The navigation process starts from the initial state (line 4 - Algorithm 4.3), though its incoming transitions cannot be analysed in the first iteration of the navigation process (line 6 -

```

input :  $P = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$  and  $\mathcal{E}_{remove}^{int}$ 
output:  $P' = \langle \mathcal{S}', q, \mathcal{E}', \Delta', \mu' \rangle$ , markedStates, cyclicPaths

1 boolean[] markedStates; boolean[] visited;
2 int[] cyclicPaths; Stack  $\langle \langle \text{Integer}, \text{Set} \rangle \rangle$  states;
3 states.push(q);
4 while not states.isEmpty() do
5     currentStateInfo  $\leftarrow$  states.pop();
6     currentState  $\leftarrow$  currentStateInfo.first();
7     visited[currentState] = true;
8     foreach (currentState, e, s)  $\in \Delta(\text{currentState})$  do
9     if  $e \notin \mathcal{E}^{int} \vee e \notin \mathcal{E}_{remove}^{int}$  then
10         addTransition((currentState, e, s)) to  $P'$ ;
11     else
12         if currentState = s then
13             addTransition((currentState, e, s)) to  $P'$ ;
14             cyclicPaths[currentState]++;
15             continue;
16         if  $\left( \begin{array}{l} |\rho(s)| = 1 \wedge \\ s \neq q \wedge \\ \Delta(s) \neq \emptyset \wedge \\ (\mu(\text{currentState}, e, s) = 1 \vee \Delta_e(s) \cap \mathcal{E}^{interface} = \emptyset) \end{array} \right)$  then
17             collapseForward (currentState, s,  $\mu(\text{currentState}, e, s)$ );
18         else
19             addTransition ((currentState, e, s)) to  $P'$ ;
20             markedStates[s] = true;
21         if visited[s] then
22             statesVisitedUpToCurrentState  $\leftarrow$  currentStateInfo.second();
23             if s  $\in$  statesVisitedUpToCurrentState then
24                 cyclicPaths[s]++;
25             continue;
26         add s to statesVisitedUpToCurrentState
27
28     if  $s \neq q \wedge s \neq \text{currentState} \wedge \text{!visited}[s]$  then
29         states.push(  $\langle s, \text{statesVisitedUpToCurrentState} \rangle$  )
30

```

Algorithm 4.2: First Phase

Algorithm 4.3) as their source states have not yet been analysed. We therefore force the first iteration of the navigation process to only traverse the outgoing transitions of the initial state (line 25 - Algorithm 4.3), even if it had been marked for analysis. Nonetheless, the incoming transitions to the initial state are analysed as the last step of the algorithm (lines 31-35 - Algo-

```

input :  $P' = \langle S', q, \mathcal{E}', \Delta', \mu' \rangle$ ,  $\text{boolean}[] \text{ markedStates}$ ,  $\text{int}[] \text{ cyclicPaths}$ ,  $\mathcal{E}_{\text{remove}}^{\text{int}}$ 
output:  $I_P = \langle S_I, q_I, \mathcal{E}_I, \Delta_I, \mu_I \rangle$ 

1  $\text{int}[] \text{ nTimesVisited}$ 
2 Queue states
3  $\text{states.push}(q_I)$ 
4  $\text{analyseInitialState} \leftarrow \text{markedState}[q_I]$ ;
5  $\text{markedState}[q_I] \leftarrow \text{false}$ ;
6 while  $\text{not states.isEmpty}()$  do
7    $\text{currentState} \leftarrow \text{states.pop}()$ ;
8   if  $\text{markedState}[\text{currentState}]$  then
9      $\rho_{\text{agg}} \leftarrow \text{Dictionary}(s, \{(s, e, \text{currentState}) \in \Delta' \mid s \in p_s(\text{currentState})\})$ 
10     $\mu_{\text{agg}} \leftarrow \text{Dictionary}(s, \sum \mu(s, e, cs)), s \in p_s(\text{currentState})$ 
11    if  $|p_s(\text{currentState})| = 1$  then
12       $s \leftarrow \text{single predecessor state}$ 
13      if  $\rho_{e\text{-agg}}(s) \subseteq \mathcal{E}_{\text{remove}}^{\text{int}} \wedge s \neq \text{currentState}$  then
14        if  $\left( \Delta(\text{currentState}) \neq \emptyset \wedge \left( \mu_{\text{agg}}(s) = 1 \vee \Delta_e(\text{currentState}) \cap \mathcal{E}^{\text{interface}} = \emptyset \right) \right)$  then
15           $\text{collapseForward}(s, \text{currentState}, \mu_{\text{agg}}(s))$ 
16        else
17          if  $\Delta_s(s) = \{\text{currentState}\} \wedge \left( \mu_{\text{agg}}(s) = 1 \vee \rho_e(s) \cap \mathcal{E}^{\text{interface}} = \emptyset \right)$ 
18            then
19               $\text{collapseBackward}(\mu_{\text{agg}}(s), s)$ 
20        else
21          foreach  $s \in p_s(\text{currentState})$  do
22            if  $\left( \rho_{e\text{-agg}}(s) \subseteq \mathcal{E}_{\text{remove}}^{\text{int}} \wedge s \neq \text{currentState} \wedge \left( \Delta_s(s) = \{\text{currentState}\} \wedge \left( \mu_{\text{agg}}(s) = 1 \vee \rho_e(s) \cap \mathcal{E}^{\text{interface}} = \emptyset \right) \right) \right)$  then
23               $\text{collapseBackward}(\mu_{\text{agg}}(s), s)$ 
24          foreach  $(\text{currentState}, e, s) \in \Delta(\text{currentState})$  do
25            if  $\text{currentState} \neq s$  then
26               $\text{nTimesVisited}[s]++$ 
27             $\text{continue}$ ;
28            if  $\text{nTimesVisited}[s] = (\rho(s) - \text{cyclicPaths}[s]) \wedge s \neq q$  then
29               $\text{states.push}(s)$ 
30
31 if  $\text{analyseInitialState}$  then
32    $\text{states.push}(q_I)$ ;
33    $\text{analyseInitialState} \leftarrow \text{false}$ ;
34    $\text{markedState}[q_I] \leftarrow \text{true}$ ;
35   goto 7;

```

Algorithm 4.3: Second phase

rithm 4.3), which we describe later in this Section. For each outgoing transition of the initial state, the algorithm *visits* their destination state s and increases the number of visited non-cyclic incoming transitions of state s (lines 26-27 - Algorithm 4.3). Thereafter, the destination state s is added to the queue of states if the following conditions hold (line 28 - Algorithm 4.3):

- all the incoming transitions to state s have been visited by the algorithm, apart from cyclic paths that start and end at state s ;
- the destination state is not the initial state.

While the second condition ensures that only a single full navigation is performed on the input PCA model, the first condition $nVisits[s] = \rho(s) - cyclicPaths[s]$ ensures that navigation progresses before cyclic paths are analysed by the algorithm as these cannot be traversed until the outgoing transitions for state s have been considered.

When the algorithm considers a state for analysis (line 8 - Algorithm 4.3), it only examines its incoming transitions if the state has been previously marked during the first phase of the algorithm (line 9 - Algorithm 4.3), *i.e.* if there is at least one incoming transition to the *currentState* labelled with an internal action in $\mathcal{E}_{remove}^{int}$. If the state has been marked for analysis, the following two steps are performed before the incoming transitions can be considered:

- the incoming transitions are aggregated in a Dictionary ρ_{agg} that indexes the transitions by their source state (line 10 - Algorithm 4.3);
- the probabilities of incoming transitions are aggregated by the source state in another Dictionary μ_{agg} (line 11 Algorithm 4.3).

When all the incoming transitions to the current state come from the same source state s (line 12 - Algorithm 4.3), then the algorithm first tries to propagate forward their aggregated probability to the outgoing transitions of the *currentState* (line 16 - Algorithm 4.3) according to the conditions defined for forward propagation (lines 13-16 - Algorithm 4.3). If the aggregated probability cannot be propagated forward, then algorithm tries to propagate it backwards to the incoming transitions of the source state s according to the conditions for backward propagation (lines 18-19 - Algorithm 4.3).

On the other hand, if the incoming transitions of the current state come from different source states (line 21 - Algorithm 4.3), then each group of transitions indexed by the source state s is analysed separately. For each group, the transitions are deleted and their aggregated probability

```

1 foreach  $(s', e', s'') \in \Delta(s')$  do
2 change  $(s', e', s'')$  to  $(s, e', s'')$   $\mu(s, e', s'') = \mu(s', e', s'') \times \mu_{cf}$ 

```

Algorithm 4.4: collapseForward(s, s', μ_{cf})

propagated to the incoming transitions of source state s if they verify the previously defined conditions for backward propagation. Note that as each group is analysed independently, even if the transitions of one group cannot be deleted it does not prevent the transitions from other groups from being removed.

```

1 foreach  $(s'', e, s) \in \rho(s)$  do
2 change  $(s'', e, s)$  to  $(s'', e, s')$   $\mu'(s'', e, s') = \mu'(s'', e, s) \times \mu_{cb}$ 

```

Algorithm 4.5: collapseBackward(s, s', μ_{cb})

After all states have been visited and all incoming transitions to **marked** states have been analysed, the incoming transitions of the initial state can now be considered for deletion if the initial state had been **marked** for analysis during the first phase (lines 31-35 - Algorithm 4.3). To this end, the initial state q_I is added to the queue of states, **marked** for analysis and the reduction process is applied to the incoming transitions of the initial state. Although the same operations and conditions are applied for the second analysis of the initial state, the navigation process does not continue after analysing its incoming transitions as the destination states s of outgoing transitions of the initial state are not pushed to the queue as the following condition is verified: $nVisits[s] > (\rho(s) - cyclicPaths[s])$. Note that the variable *analyseInitialState* needs to be set to *false* before executing the **goto** statement to ensure the algorithm does not enter into an endless loop.

4.1.2. Examples

We first illustrate a full execution of the reduction algorithm using the example in Figure 4.4.

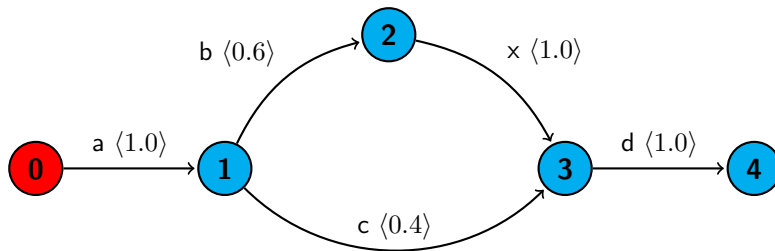


Figure 4.4.: Example PCA Model for Reduction Algorithm

Consider the application of the reduction algorithm to the PCA model in Figure 4.4 based on the following set of actions to be removed : $\mathcal{E}_{remove}^{int} = \{b, c, x\}$. The algorithm starts by considering the outgoing transitions of the initial state. Given that the action label a is not in the set $\mathcal{E}_{remove}^{int}$, the transition $0 \xrightarrow{a, 1.0} 1$ is kept in the intermediate model and the state 1 is added to the queue to be considered by the algorithm. Although the action label c is included in the set $\mathcal{E}_{remove}^{int}$, the $1 \xrightarrow{c, 0.4}$ cannot be removed by the algorithm as state 3 has other incoming transitions which may not have been fully reduced. Nonetheless, state 3 is marked for analysis during the second phase. On the other hand, the transition $1 \xrightarrow{b, 0.6} 2$ is propagated forward to transition $2 \xrightarrow{x, 1.0} 3$ as state 2 does not have other incoming transitions, x is an internal action and b is in the set $\mathcal{E}_{remove}^{int}$. In addition, state 2 is added to the queue of states so that its outgoing transitions can be analysed. The algorithm then continues by analysing the outgoing transitions of state 3 which cannot be removed as they are not labelled with actions in the set $\mathcal{E}_{remove}^{int}$. Finally, when analysing the transition $2 \xrightarrow{x, 0.6} 3$ the algorithm cannot remove it from the intermediate model and propagate its probability forward as state 3 has other incoming transitions which also need to be analysed before the transition $2 \xrightarrow{x, 0.6} 3$ can be removed. Given that no further reduction can be performed, the first-phase terminates and returns the intermediate model in Figure 4.5, with state 3 marked for analysis.

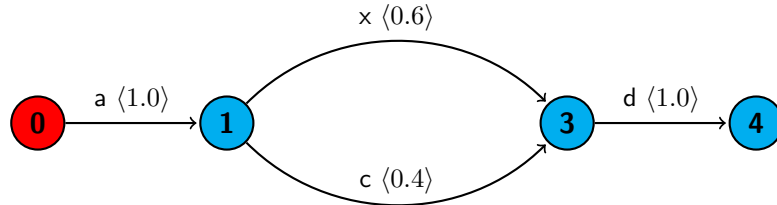


Figure 4.5.: Example PCA Model for Reduction Algorithm - Output of First Phase

The second phase starts at the initial state 0 but continues the navigation until state 3 as state 1 is not marked for analysis. When analysing the incoming transitions to state 3, the algorithm decides to remove them and propagate their probabilities to the outgoing transitions of state 3 as its incoming transitions come all from the same state and are all labelled with transitions in the set $\mathcal{E}_{remove}^{int}$. Given that no other state was marked for analysis, the algorithm finishes the second phase and produces the reduced PCA model in Figure 4.6.

In the next paragraphs we consider a series of small examples to illustrate the application of the reduction algorithm to particular scenarios.

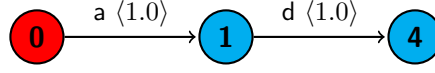


Figure 4.6.: Example PCA Model for Reduction Algorithm - Reduced Model

Forward Propagation of Single Transitions

We start by recalling the conditions for forward propagation of transitions (s, a_i, s') :

$\#(\rho_e(s')) \subseteq \mathcal{E}_{remove}^{int}$ (Incoming transitions are labelled with actions in the set $\mathcal{E}_{remove}^{int}$)

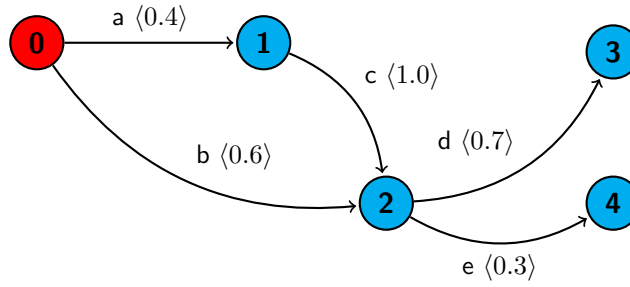
$\#(\rho_s(s')) = 1$ (Incoming transitions to state s' come from the same source state)

$\Delta(s) \neq \emptyset$ (Existence of transitions for forward propagation)

$\Delta_e(s') \cap \mathcal{E}^{in} = \emptyset$ (Invalid propagation to input actions)

$s' \neq q$ (Invalid propagation to outgoing transitions of initial state q)

$\Delta_e(s') \cap \mathcal{E}^{out} = \emptyset$ (Invalid propagation to output actions)

Figure 4.7.: Example PCA Model A_1 for Forward Propagation

The second condition ensures that the reachability properties of other paths in the input model are preserved by preventing the deletion of a single transition $s \xrightarrow{a} s'$ when state s' has other incoming transitions. For instance, in the example PCA model A_1 in Figure 4.7, the algorithm cannot delete the transition $0 \xrightarrow{b,0.6} 2$ and propagated its probability to transitions $2 \xrightarrow{d,0.7} 3$ and $2 \xrightarrow{e,0.3} 4$ as the reachability of paths $0 \xrightarrow{a,0.4} 1 \xrightarrow{c,1} 2 \xrightarrow{d,0.7} 3$ and $0 \xrightarrow{a,0.4} 1 \xrightarrow{c,1} 2 \xrightarrow{e,0.3} 4$ would be changed from 0.28 and 0.12 to 0.168 and 0.072, respectively.

Propagation to Input Actions

On the other hand, the fourth condition ensures that the reactive semantics of input actions are preserved. Consider another example PCA in Figure 4.8. If the probability of transitions

$0 \xrightarrow{b,0.6} 2$ is propagated to transitions $2 \xrightarrow{?d,1.0} 3$ and $2 \xrightarrow{e,1.0} 4$, it would violate the reactive semantics of input actions d (as shown in Figure 4.9(a)), where a discrete probability distribution is associated with outgoing transitions of a state labelled with the same action.

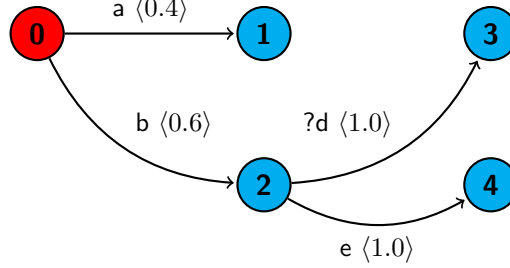


Figure 4.8.: Example PCA Model A_2 for Forward Propagation - Input Actions

Alternatively, the probability of transition $0 \xrightarrow{b,0.6} 2$ could be propagated only to the transition $2 \xrightarrow{e,1.0} 4$, thereby producing the PCA model in Figure 4.9(b). Although this model preserves both reactive and generative semantics, it does not preserve the compositional reachability properties of the original model in Figure 4.8 when composing with another model, as discussed in the next paragraphs.

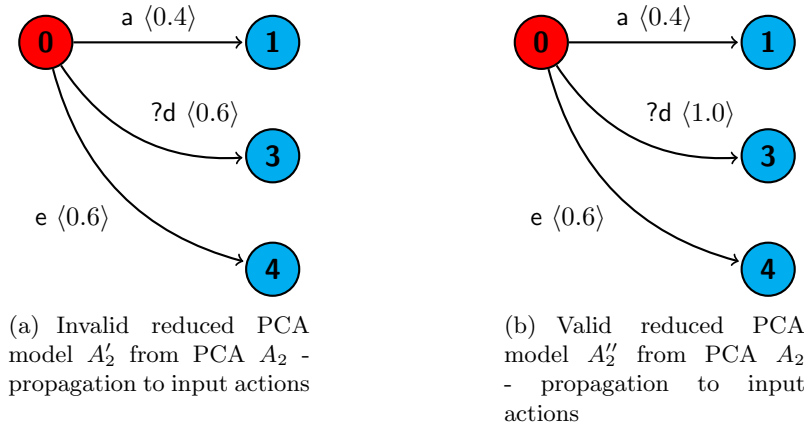
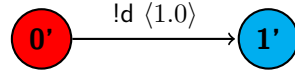


Figure 4.9.: Example Reduced PCA Models from A_2 for Forward Propagation - Input Actions

Preservation of Composition Semantics

The composite models in Figures 4.11(b) and 4.11(a) depict the composition between another PCA model B in Figure 4.10 and the original model in Figure 4.8 as well as between the model B and the reduced model in Figure 4.9(b).

Although the probability of reaching state $\langle 4, 0' \rangle$ from the initial state $\langle 0, 0' \rangle$ is preserved in the composite of Figure 4.11(b), the probability of reaching of states $\langle 3, 1' \rangle$ and $\langle 4, 0' \rangle$ is changed

Figure 4.10.: Example PCA model B - Preservation of Compositional Semantics

from 0.3 and 0.4 to 0.5 and 0.2, respectively. *Therefore, our reduction algorithm does not delete a single transition $s \xrightarrow{a} s'$ and propagates its probability forward to the outgoing transitions of state s' if at least one of these transitions is labelled with an input action.*

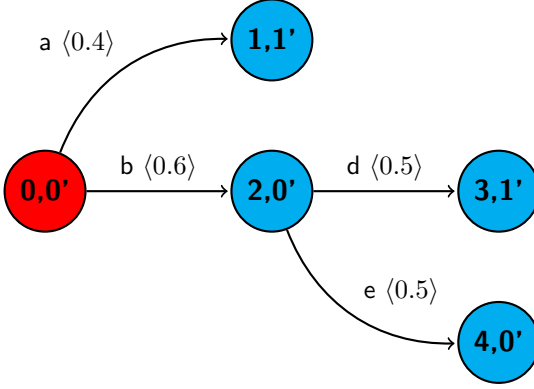
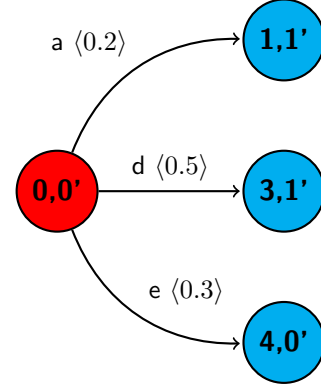
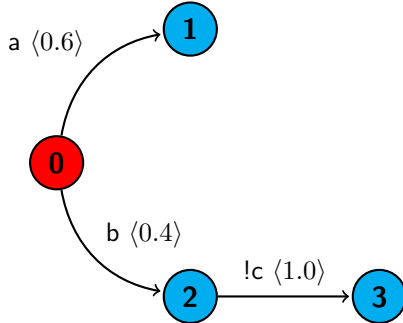
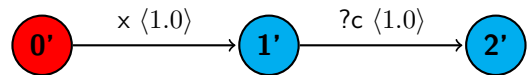
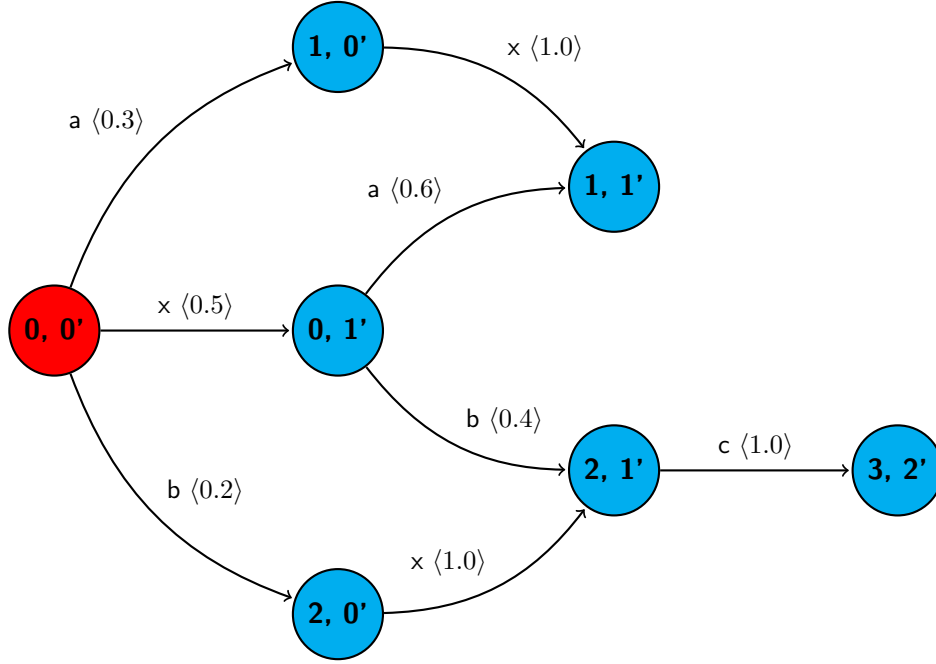
(a) Composite PCA model $A_2 \parallel B$ (b) Composite PCA model $A'_2 \parallel B$

Figure 4.11.: Example Composite Models for Preservation of Compositional Semantics

When the transition to be deleted has a probability of 1, then its propagation does not change the probabilities of outgoing actions of state s , even if these are labelled with input actions. Additionally, propagating the probability of transitions labelled with internal actions to subsequent transitions labelled with either internal or output actions does not affect the compositional properties of the reduced model if these transitions are also not labelled with output actions: $\Delta_e(s) \cap \mathcal{E}^{out} = \emptyset \vee \Delta_e(s') \cap \mathcal{E}^{out} = \emptyset$.

(a) Example PCA model C (b) Example PCA model D Figure 4.12.: Example PCA Models C and D for Preservation of Compositional Semantics


 Figure 4.13.: Composite PCA Model $C||D$ - Preservation of Compositional Semantics

Consider the example models C in Figure 4.12(a), D in Figure 4.12(b) and their composition in Figure 4.13. The probability of reaching $\langle 1, 1' \rangle$ from the initial state $\langle 0, 0' \rangle$ is given by the paths $\langle 0, 0' \rangle \xrightarrow{a, 0.3} \langle 1, 0' \rangle \xrightarrow{x, 1.0} \langle 1, 1' \rangle$ and $\langle 0, 0' \rangle \xrightarrow{x, 0.5} \langle 0, 1' \rangle \xrightarrow{a, 0.6} \langle 1, 1' \rangle$ $\left((0.3 \cdot 1.0) + (0.5 \cdot 0.6) = 0.6 \right)$, while the probability of reaching state $\langle 3, 2' \rangle$ is given by $\langle 0, 0' \rangle \xrightarrow{b, 0.2} \langle 2, 0' \rangle \xrightarrow{x, 1.0} \langle 2, 1' \rangle \xrightarrow{c, 1.0} \langle 3, 2' \rangle$ and $\langle 0, 0' \rangle \xrightarrow{x, 0.5} \langle 0, 1' \rangle \xrightarrow{b, 0.4} \langle 2, 1' \rangle \xrightarrow{c, 1.0} \langle 3, 2' \rangle$ $\left((0.2 \cdot 1.0 \cdot 1.0) + (0.5 \cdot 0.4 \cdot 1.0) = 0.4 \right)$.

Consider the reduced model C' by removing the transition $0 \xrightarrow{b, 0.4} 2$ from C and propagating its probability to the transition $2 \xrightarrow{!c, 1.0} 3$. If the model C' in Figure 4.14(a) is composed with model D , the probability of reaching states $\langle 1, 1' \rangle$ and $\langle 3, 2' \rangle$ is not preserved in the composite model depicted in Figure 4.14(b). For example, the probability of reaching state $\langle 1, 1' \rangle$ is given by the paths $\langle 0, 0' \rangle \xrightarrow{x, \frac{1.0}{1.6}} \langle 0, 1' \rangle \xrightarrow{a, 0.6} \langle 1, 1' \rangle$ and $\xrightarrow{a, \frac{0.6}{1.6}} \langle 1, 0' \rangle \xrightarrow{x, 1.0} \langle 1, 1' \rangle$, hence $\frac{1.2}{1.6} = 0.75$ which is different than the probability of reading state $\langle 1, 1' \rangle$ in the composite model of (Figure 4.13) corresponding to the composition between the original model C and the PCA model D . As transitions labelled with output actions can be blocked by another model, the reduction algorithm cannot change how their probability is normalised.

Propagation of Multiple Transitions

We now illustrate the backward propagation of multiple transitions from multiple source states, as forward propagation of multiple transitions from the same source state is equivalent to the

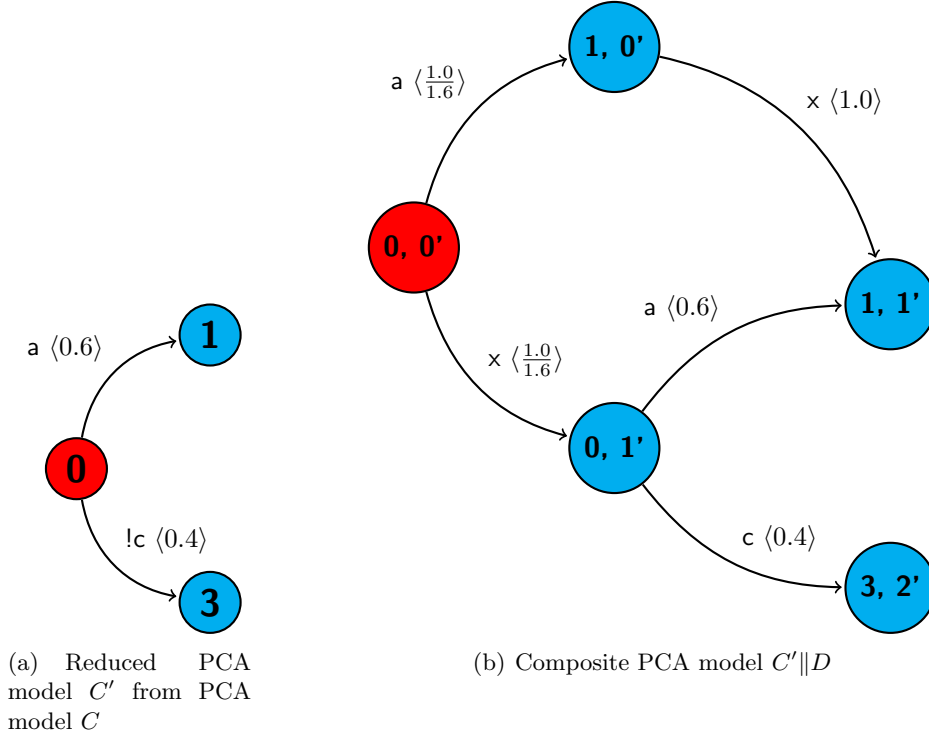


Figure 4.14.: Reduced PCA Models C' and Composite Model $C' \parallel D$ for Preservation of Compositional Semantics

case of propagating the probability of a single transition. Consider the example PCA G in Figure 4.15.

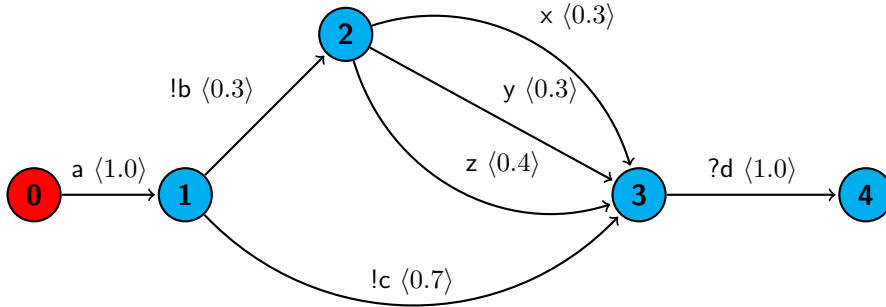


Figure 4.15.: Example PCA Model G - Propagation of Multiple Transitions from Multiple Source States

We obtain the reduced model G' in Figure 4.16 by removing the transitions $2 \xrightarrow{x,0.3} 3$, $2 \xrightarrow{y,0.3} 3$ and $2 \xrightarrow{z,0.4} 3$ from the original model G and propagating their aggregated probability to the incoming transitions of state 2, therefore preserving the reachability of paths that involve state 3, namely $0 \xrightarrow{a,1.0} 1 \xrightarrow{!b,0.3} 3 \xrightarrow{?d,1.0} 4$ and $0 \xrightarrow{a,1.0} 1 \xrightarrow{!c,0.7} 3 \xrightarrow{?d,1.0} 4$.

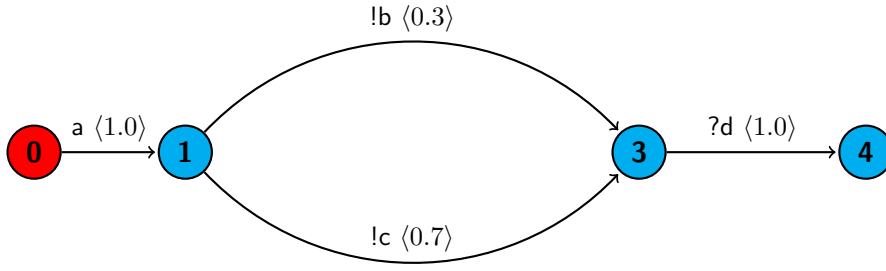


Figure 4.16.: Example Reduced PCA Model G' - Propagation of Multiple Transitions from Multiple Source States

On the other hand, the transitions $1 \xrightarrow{x,0.3} 3$, $1 \xrightarrow{y,0.3} 3$ and $1 \xrightarrow{z,0.1} 3$ of the example PCA H in Figure 4.17 cannot be propagated forward as the outgoing transition of state 3 ($3 \xrightarrow{?d,1.0} 4$) is labelled with an input action, and cannot also be propagated backwards as it would change the probability of reaching state 2 from the initial state through path $0 \xrightarrow{a,1.0} 1 \xrightarrow{c,0.3} 2$. The transition must therefore be kept.

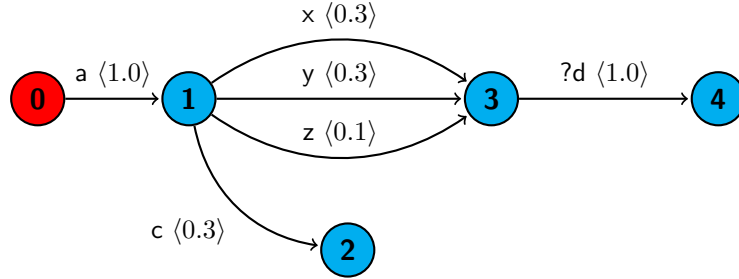
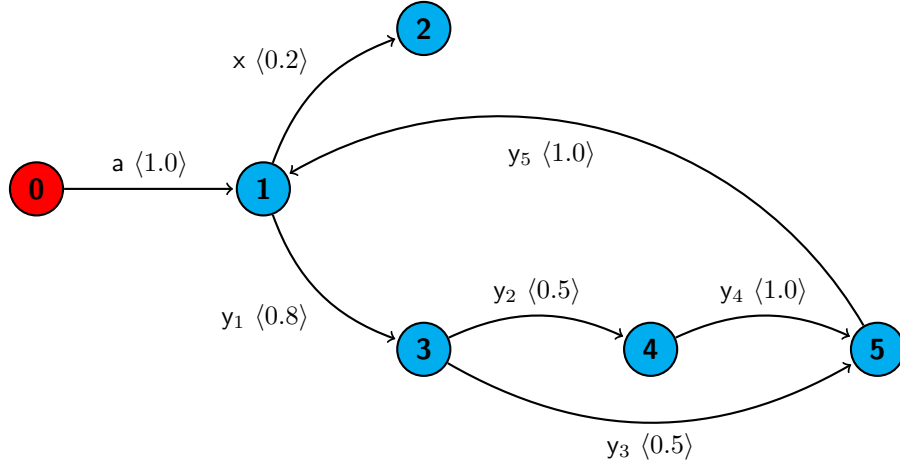
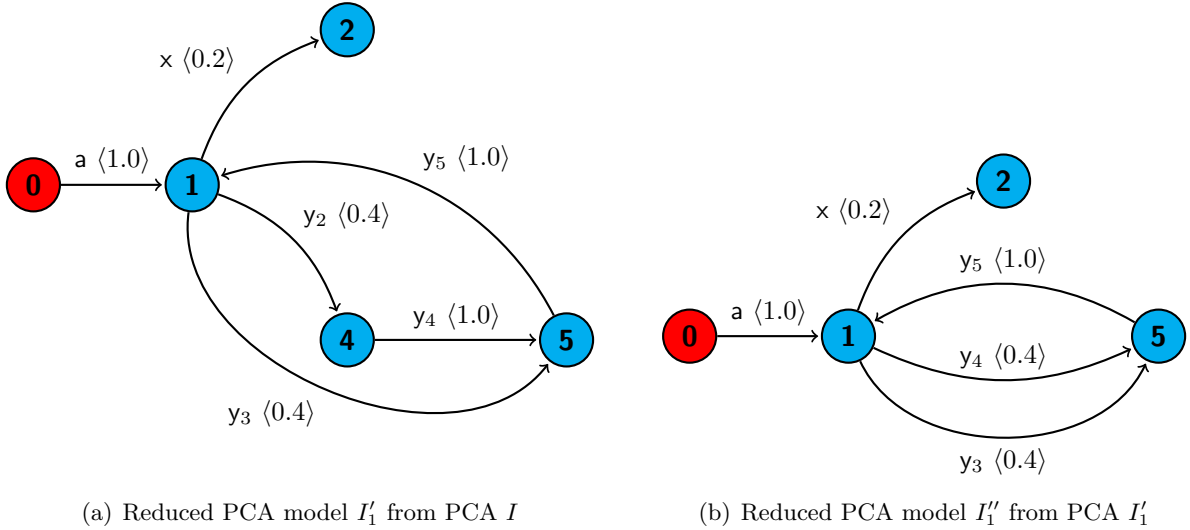


Figure 4.17.: Example Reduced PCA Model H - Invalid Propagation of Multiple Transitions

Reduction of Cyclic Paths

In the second phase of the algorithm a state s is considered to be ready for analysis when it has been visited by all its incoming transitions, apart from cyclic paths that start at that state s . Nonetheless, these paths are reduced in subsequent iterations of the algorithm. We describe in the following paragraphs how the example PCA model I in Figure 4.18, which includes cyclic paths, is reduced by the two phases of the algorithm based on the action set $\mathcal{E}_{remove}^{int} = \{x, y_1, y_2, y_3, y_4, y_5\}$.

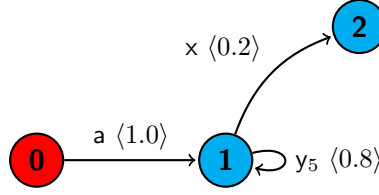
In the first phase the single transitions $1 \xrightarrow{y_1,0.8} 3$ and $3 \xrightarrow{y_3,1.0} 4$ are deleted, as these are the only incoming transitions to state 3 and 4. The PCA models in Figures 4.19(a) and 4.19(b) correspond to the intermediate reduced models after the deletion of those transitions. In the


 Figure 4.18.: Example PCA Model I - Reduction of Cyclic Paths

 Figure 4.19.: Reduced PCA Model I' after Executing First Phase on Model I - Reduction of Cyclic Paths

second phase, the transitions $1 \xrightarrow{y_4, 0.4} 5$ and $1 \xrightarrow{y_3, 0.4} 5$ are deleted when analysing the incoming transitions of state 5 as these verify the properties for deletion and forward propagation, namely:

- the transitions consist of all the incoming transitions to state 5 and they originate from the same source state 1;
- the outgoing transitions of state 5 are not labelled with input actions.

The final reduced model is shown in Figure 4.20. Note that the cyclic path has been reduced to a cyclic transition $1 \xrightarrow{y_5, 0.8} 1$. This corresponds to the best case scenario for reducing cyclic paths as loop transitions cannot be further deleted by the reduction algorithm.


 Figure 4.20.: Final Reduced PCA Model I'_2 from PCA I - Reduction of Cyclic Paths

4.2. Evaluation

After having described in detail all the steps performed by the two phases of the reduction algorithm, we evaluate in Section 4.2.1 its time complexity and illustrate the reductions in size of composite models that can be achieved in Section 4.2.2.

4.2.1. Time Complexity

We given an upper bound for the worst case time complexity of the reduction algorithm for a generic input PCA $B = (\mathcal{S}, q, \mathcal{E}, \Delta, \mu)$ to be reduced w.r.t. the set of actions $\mathcal{E}_{remove}^{int} \subseteq \mathcal{E}_B^{int}$. The time complexity of each phase is determined by the time complexity of the navigation algorithm used and the time complexity of the steps conducted at each state for analysing and deleting transitions labelled with internal actions in the set $\mathcal{E}_{remove}^{int}$. Given that both depth-first and breadth-first algorithms have the same time complexity $(\#(\mathcal{S})) + (\#(\Delta))$ when transitions are represented using an *adjacency list*, we define the time complexity of both phases as:

$$O\left(\#(\mathcal{S}) + \#(\Delta) + \#(\mathcal{S}) \cdot \text{cost}(\text{Analysis} + \text{Deletion})\right).$$

In the first phase, the time complexity for checking the conditions for deleting and propagating the probability of a deleted single transition $currentState \xrightarrow{e_{int,p}} s$ to the outgoing transitions $\Delta(s)$ of its destination state s is defined as follows:

- the conditions $|\rho(s)| = 1$ and $\Delta(s) \neq \emptyset$ have constant time complexity as it consists in verifying the size of an auxiliary data structure created when the PCA representation is instantiated;
- the conditions $\Delta(s) \neq \emptyset$, $\Delta_e(s) \cap \mathcal{E}^{out} = \emptyset$ and $\Delta_e(s) \cap \mathcal{E}^{in} = \emptyset$ take $\#(\Delta(s))$ steps to execute;
- deleting the transition and propagating its probability to outgoing transitions of state s takes $\#(\Delta(s))$ steps to complete.

Therefore, the cost of analysing and deleting transitions in the first phase is given by

$$O\left(\#(\Delta(s)) + \#(\Delta(s))\right).$$

In the navigation process of the second phase, if a state has been marked for analysis then two dictionary ρ_{agg} and μ_{agg} are created before analysing the cases for deleting transitions. Given that the number of keys of the dictionaries is known in advance $\left(\#(\rho_s(currentState))\right)$, creating these dictionaries using hashmap implementations takes $\#(\rho(currentState))$ to complete. The algorithm then considers two cases for analysis. When the incoming transitions to *currentState* come all from the same source state ($\#(\rho_s(currentState)) = 1$), then the following steps are applied:

- evaluating the condition $\rho_{e-agg} \subseteq \mathcal{E}_{remove}^{int}$ can be verified when constructing ρ_{agg} and μ_{agg} , hence we assume it has a constant time complexity;
- the conditions $\Delta(currentState) \neq \emptyset$ and $\Delta_e(currentState) \cap \mathcal{E}^{in}$ take $\#(\Delta(currentState))$ steps to execute;
 - if the conditions are valid, deleting the transition and propagating its probability to outgoing transitions of state s takes $\#(\Delta(currentState))$ execution steps;
- the conditions $\Delta_s(s) = \{currentState\}$ and $\rho_e(s) \cap \mathcal{E}^{in} = \emptyset$ are verified when the transition cannot be *collapsed forward* and take $\#(\Delta(s))$ and $\#(\rho(s))$ steps to complete, respectively;
 - deleting the transition and propagating its probability to incoming transitions of state s takes $\#(\rho(s))$ execution steps;

The time complexity associated with the first case is then given by:

$$O\left(\#(\Delta(currentState)) + \#(\Delta(currentState)) + \#(\Delta(s)) + \#(\rho(s)) + \#(\rho(s))\right).$$

On the other hand, if the incoming transitions to the *currentState* come from different source states ($|\rho_s(currentState)| > 1$), then the following steps are applied to each group of transitions indexed by their source state s :

- evaluating the condition $\rho_{e-agg}(s) \subset \mathcal{E}_{remove}^{int}$ can be verified when constructing ρ_{agg} and μ_{agg} , hence we assume it has a constant time complexity;

- the conditions $\Delta_s(s) = \{currentState\}$ and $\rho_e(s) \cap \mathcal{E}^{in} = \emptyset$ take $\#(\Delta(s))$ and $\#(\rho(s))$ steps to be verified, respectively;
- deleting the transition and propagating its probability to incoming transitions of state s takes $\#(\rho(s))$ execution steps.

The time complexity associated with the second case is then given by:

$$O\left(\#(\rho_s(currentState)) \cdot \left(\#(\Delta(s)) + \#(\rho(s)) + \#(\rho(s))\right)\right).$$

The time complexity of the reduction algorithm is defined by the time complexity of the two phases. Given that the number of incoming and outgoing transitions of each state s is not uniform, we replace in the previous time complexity formulae $\#(\rho(s))$ and $\#(\Delta(s))$ by $\bar{\rho}$ and $\bar{\Delta}$ denoting the average incoming and outgoing transitions for all the states in the model. The time required to execute the reduction algorithm can then be defined by:

$$\#(\mathcal{S}) + \#(\Delta) + \#(\mathcal{S}) \cdot \left(\underbrace{2 \cdot \bar{\Delta}}_{\text{first phase}} + \underbrace{\bar{\rho} + \max(3 \cdot \bar{\Delta} + 2 \cdot \bar{\rho}, \bar{\rho} \cdot \bar{\Delta} + 2 \cdot \bar{\rho}^2)}_{\text{second phase}} \right),$$

and the corresponding asymptotic time complexity given by

$$O\left(\#(\mathcal{S}) + \#(\Delta) + \#(\mathcal{S}) \cdot \left(\bar{\Delta} + \bar{\rho} + \bar{\rho} \cdot \bar{\Delta} + \bar{\rho}^2\right)\right).$$

Assuming the average number of incoming ($\bar{\rho}$) and outgoing transitions ($\bar{\Delta}$) of the states in the original PCA model are determined by the logarithm of the number of the total transitions ($\log(\#(\Delta))$), the time complexity of the reduction algorithm is quasilinear on the number of transitions and states of the original model:

$$O\left(\#(\mathcal{S}) + \#(\Delta) + \#(\mathcal{S}) \cdot \log^2(\#(\Delta))\right).$$

In contrast, existing state reduction techniques defined for generative systems have significantly higher time complexity. For instance, Baier and Hermans [BH97] proposed an algorithm to minimise a generative PLTS by removing transitions labelled with the *silent action* τ . The algorithm starts with an initial partition of states $\mathcal{X}_{init}$, *i.e.* a set of state equivalent classes $\{C_1, \dots, C_n\}$, such that $\cup_i C_i = \mathcal{S}$ and C_i are pairwise disjoint. This partition is successively refined by a *splitter* by finding pairs of states belonging to one equivalent class C_i for which the

probability of reaching states in other equivalence classes through $\left(s(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* s' \right)$ which denotes a sequence of zero or more non-observable transitions labelled with the internal action τ , followed by a transition labelled with action $a \in \mathcal{E}$ and succeed by zero or more non-observable transitions labelled with the internal action τ . In the worst case $n = \#(\mathcal{S})$ refinement steps need to be performed before the refinement algorithm cannot further refine the equivalence classes $\{C_1, \dots, C_n\}$. On top of that, and calculating the probability over a path may require solving a linear equation system with n variables and n equations as it may require information about the whole model [Che80]. As a result, the time complexity of this minimisation/reduction algorithm for generative PLTS systems is cubic in the number of states [BH97].

4.2.2. Empirical Evaluation

We now consider two example systems that resemble the structure of standard Web-Server applications. We analyse composite models with increasing number of clients to estimate the gains in model size reduction and associated reduction in the execution time for reliability analysis when using the reduction algorithm to compute the overall system representation. In the first system, the Server is a composite component that includes a Web-Server backend sub-component which handles requests from Clients and interacts with other backend sub-components (a Cache and a Database) to obtain the requested content, as depicted by Figure 4.21. The second system

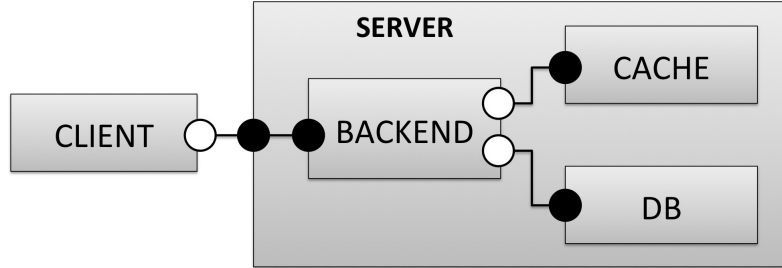


Figure 4.21.: Web-Server Architectural Configuration

(Web-Server2), whose architectural configuration is shown in Figure 4.22, is an extension of the first system that includes a backup server which is used when the main Server fails to send the requested content.

We report in Table 4.1 the results for the first Web-Server system when using from 2 to 6 Clients and for the second Web-Server system from 2 to 4 clients. Each row contains the name of the system, the original size of the composite automaton that represents the overall system behaviour, the size of the composite automaton when reduction is used, the total time

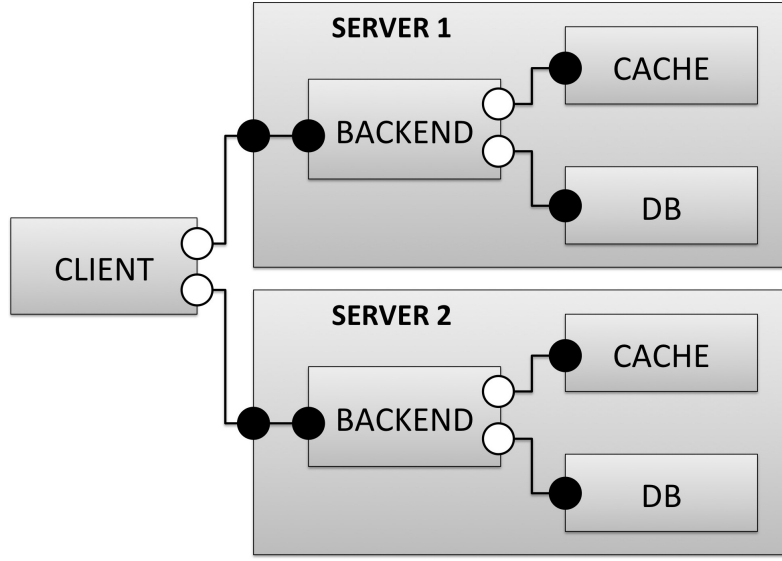


Figure 4.22.: Web-Server2 Architectural Configuration

to analyse a system using the non-reduced and the reduced representations. The execution time is then split into the time it takes to build the composite representation and the time it takes to analyse the system reliability in PRISM model checker [KNP] using the PCTL formula $1 - \mathcal{P}_{=?}[s = \text{fail} \cup \neg(s = \text{finish})]$. The column *No Reduct.* denotes the time to compute the composite representation without using the reduction algorithm, while the column *Reduct.* represents the time to reduce the component representations and then construct the composite system model using the reduced representations. Each composite PCA is then translated to a DTMC representation for analysis of reliability properties in PRISM, as these models do not contain input actions. The analysis time obtained from PRISM includes the time to construct the internal data structures in PRISM from the DTMC representation, as well as the time to analyse the overall system reliability, though the time to translate a closed PCA to the corresponding DTMC specification in PRISM is negligible. The results reported hereafter have been collected on a Macbook Pro with the following specifications: 2.8 GHz Intel Core i7, 8 GB 1600 MHz DDR3, 256GB SSD.

We first note that the composite models obtained after applying the reduction algorithm are considerably smaller despite the fact that it is not always possible to reduce all the transitions in order to preserve the probabilistic semantics. The reduction is particularly significant for larger models where the number of interleaved transitions associated with concurrent execution of different components is exacerbated. For smaller models, *e.g.* Web-Server with 2 and 3 clients and 1 server, applying the reduction algorithm and then composing requires broadly the same

Name	Original Size		Reduced Size		Execution Time			
	States	Transitions	States	Transitions	No Reduct.	Analysis	Reduct.	Analysis
Web-Server (2 Clients)	104	224	12 (88%)	28 (88%)	0.029s	0.031s	0.028s (3%)	0.002s (93%)
Web-Server (3 Clients)	304	808	36 (88%)	96 (88%)	0.035s	0.446s	0.032s (8%)	0.013s (97%)
Web-Server (4 Clients)	800	2528	91 (89%)	283 (88%)	0.041s	3.792s	0.031s (25%)	0.049s (98.7%)
Web-Server (5 Clients)	1984	7264	218 (89%)	778 (89%)	0.062s	32.564	0.042s (32%)	0.296s (99.1%)
Web-Server (6 Clients)	4736	19712	505 (89%)	2041 (90%)	0.125s	298.327	0.055s (56%)	1.724 (99.42%)
Web-Server2 (2 Clients)	696	2062	75 (89%)	212 (90%)	0.042s	2.67	0.029s (31%)	0.044s (98%)
Web-Server2 (3 Clients)	5552	19208	184 (96%)	636 (96%)	0.125s	314.6s	0.039s (56%)	1.439 (99.8%)
Web-Server2 (4 Clients)	44743	174280	2176 (95%)	8960 (94%)	0.844s	> 600s	0.082s (90%)	51.6s (> 99.99%)

Table 4.1.: Reduction Algorithm Evaluation Results - Web-Server Systems 1 and 2

amount of time as composing non-reduced models. When the number of client components is increased to 4, 5 and 6, the time to compute the composite representation is reduced by 25%, 32% and 56%, respectively. In fact, since the reduction algorithm collapses internal transitions of the composite Server component that cause the state-explosion in the non-reduced composite representation, the time to reduce the representation of the Client and Server components and then compute the composite representation barely changes when the number of clients was increased. Consistent with the reduction in size of the overall model, the analysis time in PRISM is remarkably reduced for larger systems. In particular we achieve a reduction of 99.42% when a system with 6 Clients is analysed. The results obtained with a more complex system (Web-Server2) indicate the effectiveness of the reduction algorithm in different scenarios. In fact, for a system with 2 Servers and 4 Clients we achieve a reduction of more than 99.99%. We do not provide the exact figure for the analysis time in PRISM associated with this system as we aborted the analysis after 10min, before PRISM was able to finish the construction of the internal data structures for analysing the DTMC model of the Web-Server system. These results show the usefulness of the reduction algorithm as an effective technique for scalable reliability analysis of component-based systems.

4.3. Formal Verification

In the previous Section we have empirically verified, that the models produced by the reduction algorithm preserve the properties of the original model. In order to show that the properties are preserved in the general case for any input model, we have to show that the behaviour of the reduced model is equivalent to the one of the original model when abstracting from internal behaviour. In the context of automata based models, formal verification of the reduction algorithm consists in showing that the reduced model is *weakly* bisimilar to the original input

model. This corresponds to showing that the behaviour of the reduced model is equivalent to the one of the original model when abstracting from internal behaviour, which the algorithm tries to remove. In the next paragraph we summarise the steps required for the formal verification of the reduction algorithm, while the full details are discussed in Appendix A.

Formally, *weak bisimilarity* is defined based on a *bisimulation relationship* $\mathcal{R}$ that defines a mapping between states which are considered equivalent, according to some properties [Hil96]. These properties are specific to both the behavioural equivalence and the model to which they are applied. We extend existing notions of both strong and weak bisimulation for non-probabilistic and probabilistic versions of Labelled Transitions Systems to define the conditions for strong and weak bisimulation for PCA models. To verify that the reduced PCA model is weakly bisimilar to the input PCA model, we first formalise each case of the reduction algorithm described in the previous sections as a transformation T_i which is applied to an arbitrary state of an original model A . We then demonstrate that each transformation T_i , when applied to a model A produces a weakly bisimilar model based upon a bisimulation relationship $\mathcal{R}_{T_i}$. We then show that a sequence of transformations T_i applied to model A also produces a weakly bisimilar model. This effectively corresponds to establishing that the bisimulation relationships $\mathcal{R}_{T_i}$ are transitive. Therefore, we show that the application of any sequence of transformations applied by the reduction algorithm also produces a weakly bisimilar model. Finally, we show that if two PCA models A_1 and A_2 are weakly bisimilar as defined by an equivalence relationship $\mathcal{R}_{T_i}$ ($A_1 \mathcal{R}_{T_i} A_2$), the models resulting from the application of any operator on PCA models to A_1 and/or A_2 are also weakly bisimilar. Consequently, given an original model A_1 and an expression involving any operator supported by PCA, the model A_1 can be replaced by a weakly bisimilar model A_2 whilst preserving weak bisimulation. This corresponds to showing that the weak bisimulation is a congruence for PCA.

Alternatively, weak refinement of Abstract Probabilistic Automata (APA) [DKL⁺13] could be used for compositional reliability analysis. As weak refinement of APA is a *weaker* notion than weak bisimulation, the reduced models obtained through weak refinement could be potentially smaller than the ones we obtain using our reduction algorithm. However, weak refinement for APA requires exponential time complexity [BKLS09] while the complexity of the reduction algorithm presented in this Chapter is quasilinear.

4.4. Summary and Related Work

Although composability is paramount for supporting automatic construction of fine-grained individual generic representations for each component that can be used in any architectural configuration, the interleaved transitions associated with the concurrent execution of actions from different components hinders the use of composable formalisms for modelling and analysing the behaviour of distributed applications. To this end, the reduction algorithm described in this Chapter reduces the PCA model of each component to their interface representation by removing, when possible, transitions labelled with internal actions and propagating their probabilities to surrounding transitions, whilst preserving the reachability properties of the original model even when composed with other models. As a result, the complexity of constructing composite models is decreased by reducing the representation of each component before composing with others. Note that the reduction algorithm also allows to preserve the encapsulation of composition components in the architectural model regarding the structure of their internal sub-components. After composing the PCA representations of its sub-components, the corresponding composite PCA model for the composite component can be reduced to its interface actions, thereby removing the behaviour of its sub-components and hiding it from external components. This is similar to the encapsulation of internal bindings between sub-components that architecture models provide.

The complexity of reliability analysis is also significantly reduced by using smaller representations computed by the reduction algorithm. As the effort of reducing the size of the automaton for analysis is performed before using PRISM model checking tools, the complexity of analysing different properties on the same model is minimised. *In fact, the experimental results suggest that the gains obtained by applying the reduction algorithm before composing are much greater than the gains obtained by the state reduction techniques performed in PRISM on the DTMC model translated from the non-reduced composite model. Note that the time complexity of the reduction algorithm for the worst case is quasilinear on the number of transitions and states of the original model, these results indicate that the time complexity of reduction techniques in PRISM is greater than the time complexity of our reduction algorithm.* We also reduce the PCA models at each level of a composition hierarchy, *i.e.* before and after constructing composite representations, with the aim of limiting the size of composite representations linear in the number of states of individual models. This also allow us to preserve representations of sub-parts of

the system which can be re-used when other parts are adapted. In contrast, the state reduction techniques in PRISM can only be applied to the final DTMC model of the system.

Our algorithm is also more efficient than existing state reduction techniques defined for generative systems [BH97] that involve a recursive refinement which requires analysis of the full model at each iteration. In contrast, our reduction algorithm for PCA models performs an iterative navigation process and deletes incoming/outgoing transitions to a state when their deletion does not affect the reachability properties of surrounding states and transitions. While the algorithm defined by Baier and Hermains [BH97] cannot be applied to PCA models as it does not cater for reactive semantics associated with input actions, our reduction algorithm can also be applied to generative PLTS systems as these can be seen as a special case of PCA models without input actions. Although a compositional analysis algorithm which seeks to apply CRA concepts to systems with input/output actions has been proposed before for PIOA models [SP99], the resulting automaton may be inconsistent with PIOA semantics. For example, the algorithm deletes transitions labelled with a non-observable action that lead to an absorbing state but this is not appropriate since their probabilities cannot be propagated to other transitions. Although we have empirically verified for the examples described in this Chapter that the reduced PCA models produced by our reduction algorithm preserve the reliability properties of the original models, we have not shown that this result holds for any general input model. In Appendix A we show that a generic input model and the reduced representation produced by the algorithm are equivalent according to notions of probabilistic behaviour equivalences. In the next Chapter we discuss how the work hitherto presented can be combined with existing approaches to define a multi-view framework that closely integrates architectural, behavioural and non-functional aspects of a component-based systems to automate architecture reconfiguration.

5. Automating Architectural Reconfiguration

Autonomic computing [KC03] advocates departing from traditionally integrated management functions (*e.g.* FCAPS: fault, configuration, accounting, performance and security) and moving towards locally integrated autonomous operation, though it has not addressed the integration with software design of adaptive systems. Traditionally, the design of systems and service management has been separated from the design of the software system [LDS⁺08]. For example, the analysis of quality-of-service (QoS) parameters is often based on profiling tools, which are external observers with respect to the software system. But if the software changes, there may be a considerable delay until the changes are detected. Moreover, modern systems are becoming increasingly dynamic and adaptive, particularly in the case of novel service architectures and pervasive systems (*e.g.* sensor networks, body area networks, embedded systems) where the physical system itself may rapidly evolve at run-time. In such systems a separation between the design of the management system and the design of the system itself is less obvious.

Several approaches associate non-functional parameters with components and select configuration changes based on aggregated measures of the non-functional parameters of their components. Sykes *et al.* [SHMK10] consider the components independently, Grassi *et al.* [GMM13]; extend this to account for dependencies between components. But neither consider the behavioural aspects, which are necessary to accurately assess the system's reliability. Indeed, a component's reliability depends on how and which parts of its behaviour are used. Feature driven models [EEM10, MLD⁺11a] consider alternative implementations that can be changed at run-time, though these are not composable and cannot be easily analysed. Similarly, existing approaches for reliability analysis are based on DTMC models that consider each component as a black-box. In addition, a new DTMC has to be manually defined for each architectural configuration as the DTMC model of a composite component cannot be automatically constructed from models of its sub-components.

Self-management frameworks implement variations of a MAPE loop (Monitoring, Analysis, Planning and Execution) but integration with software models (Knowledge) is also limited. The Self-Managed Cell (SMCs) [SL10] framework supports distributed and composable autonomous components through Event-Condition-Action (ECA) policies. Rainbow [GCH⁺04] integrates architectural models with ECA policies that change system parameters on the basis of their costs and benefits given as annotations. But in such frameworks changes must be anticipated in advance when policies are defined.

We build herein upon the work presented in the previous chapters and show how component reconfiguration can be automated by using a multi-view framework that closely integrates architectural, behavioural and non-functional aspects of a component-based systems. In Section 5.1 we present a general methodology that encompasses model extraction and analysis while integrating architectural, behavioural and management views of application components. We then show in Section 5.2 how reconfiguration can be achieved using a centralised method where a single node selects architectural configuration that verifies reliability requirements, *e.g.* maximising system reliability. We also describe a distributed version of the reconfiguration method where each component autonomously decides on its internal configuration and evaluate the performance of reconfiguration processes in Section 5.3.

5.1. Notational Elements

Component-based models can be reconfigured by changing the bindings between provided and required interfaces. We use component in a generic sense to denote encapsulation of behaviour and assume that components are composable, *i.e.* a composite component is realised as a configuration of sub-components. When multiple configurations achieve the desired functional behaviour, non-functional properties such as reliability are used to select the configuration to deploy. Management services monitor system behaviour, measure non-functional parameters and enact reconfiguration changes to preserve the system's reliability. Component-based applications and services can therefore be seen from an architectural, behavioural or management perspective. However, multiple perspectives of the same system may be inconsistent [FKN⁺92] so their interdependencies need to be discussed at some length.

5.1.1. Architectural View

The architectural view comprises the components, their *provided* and *required* interfaces as well as their bindings; for composite components, sub-components and internal bindings are also specified. Components provide encapsulation of behaviour and autonomy: each component can be seen as managing its internal configuration of sub-components and adapting it to achieve the required reliability, though this need not be the case, and a distinction can be drawn between autonomous and non-autonomous components if necessary. Similarly to Sykes *et al.* [SHMK10], we use the Darwin [MDEK95] notation to represent the system's architecture.

We show in Figures 5.1 and 5.2 the Darwin graphical and textual notations of Client and Server, respectively, where the Client component specifies a requirement interface r and the Server component defines a provided interface p .

```
component Client {  
  require r;  
}
```

(a) Client - Darwin Textual Representation

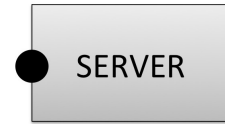


(b) Client - Darwin Graphical Representation

Figure 5.1.: Darwin Representations of Client Component

```
component Server {  
  provide p;  
}
```

(a) Server - Darwin Textual Representation



(b) Server - Darwin Graphical Representation

Figure 5.2.: Darwin Representations of Server Component

The corresponding graphical and textual representations to a composite **ClientServer** System are depicted in Figure 5.3, where the textual notations consists of a list of component instances and bindings between their provided and required interfaces.

We define for each component C $\mathcal{P}_C = \{p_1, \dots, p_n\}$ as the set of its provided interfaces and $\mathcal{R}_C = \{r_1, \dots, r_n\}$ as the set of its required interfaces.

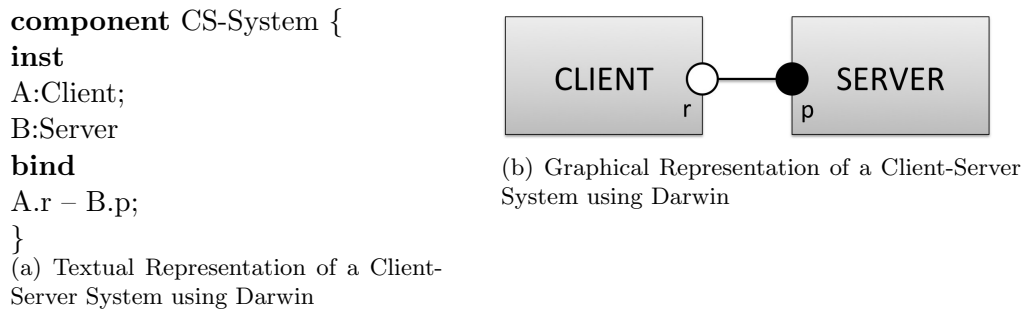


Figure 5.3.: Darwin Representations of Client-Server System

5.1.2. Behavioural View

The behavioural view comprises the representation of the components' probabilistic behaviour described using our modelling formalism PCA. Input-output actions are associated with the interfaces of a component, where the set $\mathcal{E}(C.i)$ denotes the interface actions associated with interface i of component C , where $i \in \mathcal{P}_C$ or $i \in \mathcal{R}_C$. In the case of provided interfaces, an input action models the possibility of the associated method being invoked followed by internal behaviour and an output action to denote that the method result is being returned/sent. A corresponding pair of output-input actions is associated with a required interface, where the output action designates the invocation of the required functionality and an input action indicates the component receiving the result of the invoked functionality.

The behaviour representation of composite components is automatically constructed by composing in parallel the models of its sub-components. Input and output actions are synchronised to model interactions between two components *i.e.* communication along component bindings, and internal actions are interleaved to denote their concurrent execution. Although a component's PCA model is independent from the configuration in which the component is deployed, it needs to be adjusted prior to composition in order to reflect the bindings in the architectural model. This is achieved by removing unused behaviour associated with unbound provided interfaces and by relabelling actions associated with multiple bindings to a provided interface.

While in the architectural view, the interfaces of a component hide its internal behaviour, in the behavioural view the hiding operator removes internal behaviour, thereby transforming, when possible, a component's PCA into a representation that only includes interface actions. The same principle applies to composite components that encapsulate their internal structure in the architectural view.

An underlying Interface Automata representation can be automatically extracted from both simple and composite PCA models by removing probabilities from transitions between states. This representation is then used to determine compatibility of components with bindings between provided and required interfaces.

The architectural and behavioural view are therefore not independent but closely correspond to each other. In the next Section we describe the management services required to automate architectural reconfiguration and discuss their link to the architectural and behaviour view.

5.1.3. Management View

While the architectural view represents the components, their interfaces and bindings, and the behavioural view models the probabilistic state transitions and their reachability, the management view concerns maintaining the component inventory, monitoring component execution to calculate the state transition probabilities, detecting violations of desirable properties, as well as deciding upon and performing reconfiguration.

Components can be self-managed *i.e.* implementing their own management services and managing their own internal configuration of sub-components. For the purpose of the distributed reconfiguration algorithm presented in Section 5.2.3 we consider each component as an autonomous system, *i.e.* managing the internal configuration of its sub-components. However, the management of several components, or a whole sub-hierarchy of component configurations can be ensured by a single management system. Indeed, at the other end of the spectrum, management can be fully centralised as we describe in Section 5.2.1. But whether distributed or centralised, the management view must contain at least the following elements.

Monitoring

The *Monitoring* function is responsible for updating the behavioural models relying upon execution traces to keep an accurate and consistent representation of the system execution profile. Changes in a component's execution profile must be detected and the probabilities μ associated with the transitions in the PCA model of each component must be updated. This includes the probabilities of all transitions including failures (*i.e.* transitions to the ERROR state).

Instance Inventory

Re-configuring a system requires knowing at all times which components are available and thus maintaining an inventory of available components and their characteristics. Typically, this requires discovering new components when they become available and detecting their failure or absence. The components can be services or embedded devices such as sensors whose presence can be detected at run-time. In addition, the presence of a new component does not in itself trigger reconfiguration, as such changes can occur only in response to violations of non-functional requirements such as service-level objectives. Note that functional compatibility between components interfaces is not sufficient to ensure components can be bound as their behaviour may be mismatched, *e.g.* different protocol versions. Conventional (*i.e.* non-probabilistic) analysis techniques can be applied to the underlying Interface Automata representations of two components with bindings between their interfaces to detect the presence of deadlocks or violation of pre-determined constraints.

Decision-Making

The previous management components along with the behavioural and architectural views provide the means for *Decision-Making* components to reason about the system in order to analyse the need for adaptation in order to satisfy goal and utility policies. In the next Section we detail the steps of a reconfiguration process that enables the identification of necessary architectural changes that maximise reliability.

5.2. Architectural reconfiguration

When multiple functionally equivalent architectural configurations are available, non-functional properties are used to select the most suitable configuration. PCA models are applied to automatically construct a composite representation for a given configuration based on the representation of each component. Although the calculation of reliability properties for a given configuration can be automated using the composite representation, the bindings between components determine how such representation is constructed. *We focus in the next Section on selecting the architectural configurations which maximise reliability where the system's reliability is analysed using the composite PCA representation.* We first describe a centralised solution to construct the composite model of each alternative architecture configuration (Section 5.2.1) and

then present the distributed case where each component behaves autonomously and selects the most reliable configuration of its sub-components (Section 5.2.3).

5.2.1. Centralised Algorithm

The first step of the reconfiguration algorithm consists in determining the possible architectural configurations that satisfy the functional requirements of the system. A centralised management keeps an instance inventory of available component instance $\mathcal{C} = \{C_1, \dots, C_n\}$, where each C_i is associated with a set of provided interfaces $\mathcal{P}_{C_i} = \{p_1, \dots, p_n\}$ and a set of required interfaces $\mathcal{R}_{C_i} = \{r_1, \dots, r_n\}$. An architectural configuration is then specified using Darwin based on the set of components $\mathcal{C}_{arch}$ and the bindings between their compatible provided and required interfaces. The probabilistic behaviour of each component C_i is defined by the PCA: $A_{C_i} = \langle \mathcal{S}_{C_i}, q_{C_i}, \mathcal{E}_{C_i}, \Delta_{C_i}, \mu_{C_i} \rangle$.

We then use the dependency analysis algorithm defined by Sykes *et al.* [SHMK10] to determine the set of all possible functional architectural configurations $\{ \mathcal{B}_{arch_1}, \dots, \mathcal{B}_{arch_n} \}$ from the set of available component instances $\mathcal{C}$ and functional requirements of the system. For each configuration of bindings $\mathcal{B}_{arch_i}$, if the provided interfaces of the components in $\mathcal{C}_{arch_i}$ are all bound to required interfaces of components in $\mathcal{C}_{arch_i}$, then the composite behavioural representation for $\mathcal{B}_{arch_i}$ is given by the parallel composition of the PCA models of its components: $A_{C_1} \parallel \dots \parallel A_{C_n}, A_{C_i} \in \mathcal{C}_{arch}, i \in \{1, \dots, n\}, n = \#(\mathcal{C}_{arch_i})$.

The architectural configurations produced by the dependency analysis algorithm guarantee that all functional requirements of each component are satisfied, *i.e.* all required interfaces are bound to a provided interface. However, the algorithm assumes that all required interfaces are needed even when not all provided interfaces of all components are bound as some functionality may not be used. We describe in the next paragraphs how the configurations generated by the aforesaid dependency analysis algorithm [SHMK10] can be refined to only consider the functional dependencies of the active behaviour of each component. In other words, some required interfaces of each component may not be needed as some of its provided interfaces are not bound.

For each architectural configuration $\mathcal{B}_{arch_i}$ of bindings between provided and required interfaces of a set of components, the following steps are applied to all components in $\mathcal{B}_{arch_i}$:

1. Remove behaviour associated with unbound provided interfaces:

- given set of unbound interfaces

$$unbound_{C_i} = \{p_j \in \mathcal{P}_{C_i} \mid \nexists (C_i.p_j - -C_k.r_j) \in bindings(\mathcal{B}_{arch_i})\};$$

- determine sub-set of input actions associated with unbound interfaces

$$unbound_{\mathcal{E}_{C_i}} = \bigcup_{p \in unbound_{C_i}} \mathcal{E}_{C_i}(p) \cap \mathcal{E}_{C_i}^{in};$$

- compute PCA $A_{C_i-\mathcal{B}_{arch_i}}$ denoting *active* behaviour w.r.t. $\mathcal{B}_{arch}$ using the interface ($@$) operator:

$$A_{C_i-\mathcal{B}_{arch_i}} = A_{C_i} @ unbound_{\mathcal{E}_{C_i}};$$

2. Compute interface representation $IA_{C_i-\mathcal{B}_{arch_i}}$ by removing internal behaviour:

$$IA_{C_i-\mathcal{B}_{arch_i}} = A_{C_i-\mathcal{B}_{arch_i}} \setminus \mathcal{E}_{C_i-\mathcal{B}_{arch_i}}^{int};$$

3. Update $\mathcal{B}_{arch_i}$ based upon active functional dependencies:

- determine set of active required interfaces

$$fun_{dep} = \{r \mid r \in \mathcal{R}_{C_i} \wedge \mathcal{E}_{C_i}(r) \subseteq \mathcal{E}_{IA_{C_i-\mathcal{B}_{arch_i}}}\};$$

- remove bindings to inactive required interfaces $\mathcal{R}_{C_i} \setminus fun_{dep}$ from $\mathcal{B}_{arch_i}$

4. Revise interface representation $IA_{C_i-\mathcal{B}_{arch_i}}$ of components that had provided interfaces bound to the required interfaces removed in the previous step;

- if all the bindings to provided interfaces of component C_i are removed, then C_i is removed from $\mathcal{C}_{arch_i}$;
- repeat step 4 for components that were required by C_i .

Note that determining the functional dependencies between the provided and required interfaces of each component only requires the system designer to specify the input-output actions associated with each interface, which are also necessary to adjust the PCA representation of each component according to the bindings of a particular architectural configuration. The same steps are applied to all architectural configurations generated by the dependency analysis algorithm

and the reconfiguration algorithm constructs the corresponding composite PCA model for each alternative configuration $\mathcal{B}_{arch_i}$ using the revised PCA representations $IA_{C_i-\mathcal{B}_{arch_i}}$ computed during the functional dependency analysis.

If the configuration $\mathcal{B}'_{arch_i}$ includes multiple bindings to a provided interface, we use the relabelling operator to modify the interface representation $IA_{C_i-\mathcal{B}_{arch_i}}$ of the components involved in order to support and distinguish requests from different components. The composite model corresponding to the revised configuration $\mathcal{B}'_{arch_i}$ is then obtained by composing in parallel the interface representation of each component:

$$A_{\mathcal{B}'_{arch_i}} = IA_{C_1-\mathcal{B}'_{arch_i}} \parallel \dots \parallel IA_{C_n-\mathcal{B}'_{arch_i}}.$$

Before the reliability of each configuration $\mathcal{B}'_{arch_i}$ is analysed, we extract the underlying Interface Automata representation from the composite model $A_{\mathcal{B}'_{arch_i}}$ and verify if the configuration does not lead to deadlock states due to behaviour mismatch between components in $\mathcal{C}_{arch_i}$. If the composite IA model has deadlock states, then configuration $\mathcal{B}'_{arch_i}$ is discarded by the re-configuration algorithm.

For each valid configuration, the interface representation $IA_{C_i-\mathcal{B}'_{arch_i}}$ of each of the components involved does not contain behaviour associated with unbound provided interfaces. As a result, the composite model $A_{\mathcal{B}'_{arch_i}}$ is a closed representation, *i.e.* it does not contain input actions as these have all been synchronised with the corresponding output actions. Such model is automatically translated to a corresponding DTMC representation which is then used for analysis of reliability properties in PRISM model checker. The reconfiguration algorithm then selects the valid configuration that maximises the reliability of the system. The management component(s) then instantiate the new bindings when the components are ready for reconfiguration, *e.g.* according to the tranquility criterion [VEBD07].

5.2.2. Example Web System for Centralised Reconfiguration

We describe in the next paragraphs how centralised reconfiguration is applied to a Web system example, whose architectural configuration is shown in Figure 5.4.

The Web system comprises the following components:

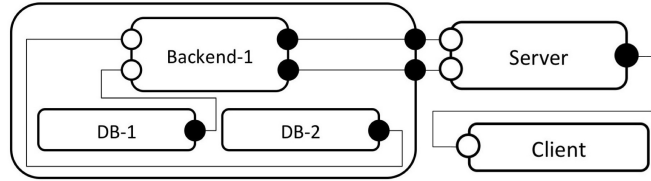
- *Client* makes requests for web pages to the

- *Server* which uses a backend infrastructure to obtain dynamic text content and images for the pages requested by the Client;
- a composite component *Backend-1* provides both text content and images relying upon two databases (*DB-1* and *DB-2*).

component $\mathcal{B}_1 = \{$

inst

S:Server;
C:Client;
B1:Backend-1;
dB1:DB1;
dB2:DB2;



bind

(b) Graphical representation of the WebSystem using Darwin

$S.getWebPage$ -- $C.getWebPage$
 $B1.getImage$ -- $S.getImage$
 $B1.getContent$ -- $S.getContent$
 $B1.getContentDB$ -- $dB1.getContentDB$
 $B1.getImageDB$ -- $dB2.getImageDB$
 }

(a) Textual representation of the Web-System using Darwin

Figure 5.4.: Description of Configuration $\mathcal{B}_1$ of the WebSystem using Darwin

This system is instantiated with the bindings configuration $\mathcal{B}_1$ in Figure 5.4. Since the configuration $\mathcal{B}_1$ does not have unbound interfaces, the corresponding composite PCA representation can be simply obtained as the parallel composition of the interface representation of each component in $\mathcal{B}_1$:

$$IA_{Client-\mathcal{B}_1} \parallel IA_{Server-\mathcal{B}_1} \parallel IA_{Backend-1-\mathcal{B}_1} \parallel IA_{DB1-\mathcal{B}_1} \parallel IA_{DB2-\mathcal{B}_1} .$$

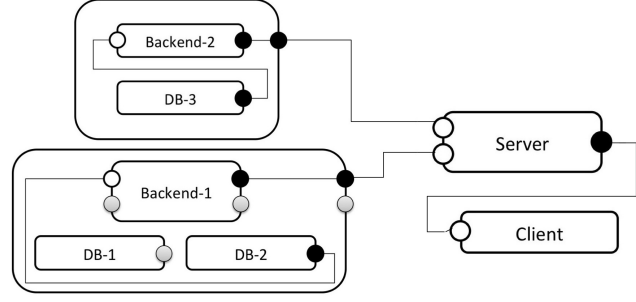
Consider that a new component *Backend-2* becomes available. As it provides images from a separate database *DB-3* an alternative configuration $\mathcal{B}_2$, depicted in Figure 5.5, can be considered for reconfiguration. In contrast with $\mathcal{B}_1$, the composite representation for $\mathcal{B}_2$ cannot be simply obtained through the parallel composition of A_{Client} , A_{Server} , $A_{Backend-1}$ and $A_{Backend-2}$ as *this would imply that the required interface $getImage$ of component *Server* is bound to (synchronised with) both *Backend-1* and *Backend-2**. Therefore, the PCA representation of *Backend-1*

```

component  $\mathcal{B}_2 = \{$ 
inst
S:Server;
C:Client;
B1:Backend-1;
B2:Backend-2;
dB1:DB1;
dB3:DB3;
bind
  S.getWebPage      -- C.getWebPage
  B2.getImage       -- S.getImage
  B2.getImageDB     -- dB3.getImageDB
  B1.getContent     -- S.getContent
  B1.getContentDB   -- dB1.getContentDB
}

```

(a) Textual representation of the Web-System using Darwin



(b) Graphical representation of the WebSystem using Darwin

Figure 5.5.: Description of Configuration $\mathcal{B}_2$ of the WebSystem using Darwin

needs to be adjusted to include only the actions associated with interfaces used in configuration $\mathcal{B}_2$:

$$I_{Backend-1-\mathcal{B}_1} = \left(A_{Backend-1} @ \mathcal{E}(getContent) \right) \setminus \mathcal{E}_{Backend-1}^{int}.$$

Given that the provided interfaces of other components in $\mathcal{B}_2$ are all bound, their interface representation is simply obtained by removing the internal actions of each model.

The composite PCA model corresponding to configuration $\mathcal{B}_2$ is then given by the parallel composition of the interface representation of each component:

$$A_{\mathcal{B}_2} = I_{Client-\mathcal{B}_2} \parallel I_{Server-\mathcal{B}_2} \parallel I_{Backend-1-\mathcal{B}_2} \parallel I_{Backend-2-\mathcal{B}_2} \parallel I_{DB1-\mathcal{B}_2} \parallel I_{DB3-\mathcal{B}_2}.$$

The algorithm then calculates the reliability of $\mathcal{B}_2$ to determine if the system needs to be reconfigured.

5.2.3. Distributed Reconfiguration

The described centralised reconfiguration process assumes a centralised management system that has knowledge about all components, their architecture, configuration and behaviour representation. However, in many cases (*e.g.* services, pervasive systems) central control may not be possible and components need to be autonomous. Although knowledge about alternative bind-

```

1   $\{\mathcal{I}_{C_u}, \mathcal{R}_{C_u}\} = receive(C_u)$ 
2   $A'_C = A_C @ unbound_{\mathcal{E}_C}; \mathcal{I}_{C-C_u} = A'_C \setminus \left\{ \mathcal{E}_C - \left( \mathcal{E}_{\mathcal{I}_{C_u}} \cup \bigcup_{j \in \mathcal{R}_C} \mathcal{E}(j) \right) \right\}$ 
3   $\mathcal{I}_C = \mathcal{I}_{C-C_u} \parallel \mathcal{I}_{C_u}$ 
4   $\mathcal{R}'_C = \emptyset$ 
5  foreach  $r \in \mathcal{R}_C$  do
6      if  $\mathcal{E}(r) \cap \mathcal{E}_{\mathcal{I}_C} \neq \emptyset$  then
7          add  $r$  to  $\mathcal{R}'_C$ 
8
9   $maxReliability = -\infty$ 
10  $\mathcal{B}_{maxReliability} = \emptyset$ 
11  $\mathcal{C}_{available} = availableComponentInstances()$ 
12 foreach  $\mathcal{B} \in permutations(\mathcal{C}_{available}, \mathcal{R}'_C)$  do
13      $\mathcal{A}_{\mathcal{B}} = \mathcal{I}_C$ 
14     foreach  $C_l \in \mathcal{B}$  do
15          $\mathcal{I}_{C-C_l} = \mathcal{I}_C \setminus \left( \mathcal{E}_C - \bigcup_{i \in \mathcal{B}(C_l)} \right)$ 
16         send  $(\mathcal{I}_{C-C_l}, C_l)$ 
17          $\mathcal{I}_{C_l} = receive(C_l)$ 
18          $\mathcal{A}_{\mathcal{B}} = \mathcal{A}_{\mathcal{B}} \parallel \mathcal{I}_{C_l}$ 
19
20      $reliability_{\mathcal{A}_{\mathcal{B}}} = reliability(\mathcal{A}_{\mathcal{B}})$ 
21     if  $reliability_{\mathcal{A}_{\mathcal{B}}} > maxReliability$  then
22          $maxReliability = reliability_{\mathcal{A}_{\mathcal{B}}}$ 
23          $\mathcal{B}_{maxReliability} = \mathcal{B}$ 
24
25 if  $\mathcal{P}_C \neq \emptyset$ 
26      $\mathcal{I}_C = \mathcal{I}_{C-C_u} \parallel \mathcal{I}_{C_{l_1}} \parallel \dots \parallel \mathcal{I}_{C_{l_n}}, \mathcal{I}_{C_{l_i}} \in \mathcal{B}_{maxReliability}$ 
27      $\mathcal{I}'_C = \mathcal{I}_C \setminus \left( \mathcal{E}_{\mathcal{I}_C} - \mathcal{E}_{\mathcal{I}_{C_u}} \right)$ 
28     send  $(\mathcal{I}'_C, C_u)$ 
    
```

Algorithm 5.1: Distributed Assembly Algorithm

ings can be exchanged between components *e.g.* using gossip algorithms [SMK11], the choice of reconfiguration needs to be performed locally. This requires them to exchange information regarding their interfaces and behaviour so that each component can analyse the reliability properties of alternative reconfiguration. We describe in the next paragraphs a distributed reconfiguration process that defines how components exchange information for selecting between alternative instances for their required functionality in order to maximise the system reliability. Note that the local choice performed at each component is based on the reachability of a generic failure state, without being specific to the execution of particular (failure) actions. Consequently, the described process for exchanging information between components is specific

to such reliability property. In contrast, the centralised approach supports the analysis of any reliability property as the central node has knowledge about the PCA models of all components.

The general algorithm is described in Algorithm 5.1 and is divided in two phases. A descendent phase enables each component C to determine which of its required interfaces are needed based on information received from upstream dependencies regarding which provided interfaces of component C are going to be used. Component C can thereafter calculate alternative configurations to the required interfaces computed in the previous step. For each alternative configuration $\mathcal{B}$, component C sends to each component in $\mathcal{B}$ which of its provided interfaces are needed and how component C would use them if configuration $\mathcal{B}$ were to be instantiated. The descendent phase continues until components without required interfaces are reached; an ascending phase starts thereafter where each component locally selects between alternative configurations to its required interfaces and sends the resulting model to its upstream dependencies until a component without provided required interfaces is reached. In the next paragraphs we describe in more detail each phase of the distributed reconfiguration process.

The descendent phase is initiated from a component C_1 with no provided interfaces, *i.e.* the root of a composition hierarchy, which then calculates all its possible bindings to its required interfaces and available component instances (Algorithm 5.1 - lines 11-12). For each alternative configuration $\mathcal{B}$, the component C_1 transmits to each sub-component the PCA representation dictating how the provided interfaces of the sub-component would be used if configuration $\mathcal{B}$ were to be instantiated (Algorithm 5.1 - lines 12-16). At each step of the descendent phase, a component receives a PCA representation from an upstream dependency and adjusts its PCA representation according to the provided interfaces required by upstream dependencies. It then determines which of its required interfaces are needed and performs the previously described steps for its downstream dependencies. The descendent phase stops when a *leaf* component is reached. An ascendant phase is then started where each component selects the configuration that maximises its reliability, and propagates a PCA model that reflects that choice upwards in the hierarchy, until the root of the composition hierarchy is reached.

These computations do not need to be repeated at each change. For example, if a component C becomes unavailable, the component(s) that used C have to select an alternative configuration to meet their functional requirements. As these components instances have calculated all the possible configurations w.r.t. to their required interfaces (Algorithm 5.1 - lines 11-22), no computation is required for downstream dependencies. However, as the new configuration

is less reliable, these component instances need to send their new reduced PCA to upstream dependencies (Algorithm 5.1 - lines 24-26), which may lead to changes in the configuration which previously produced the highest reliability. Similarly, when a new component C becomes available, existing component instances that have matching required interfaces can run the re-configuration algorithm based on the new set of available component instances ($C_{available}$). If the new configurations with the new component C produce a higher reliability, then a new reduced PCA representation is sent to upstream dependencies, which may themselves select a different configuration.

When the execution profile of a component C changes, the transition probabilities in its PCA model change accordingly. Therefore, the reliability of the possible configurations involving C may not be the same. To calculate the new reliability value for each configuration, the component C re-constructs a new composite model based on the PCA representations received from both upstream and downstream dependencies, similar to the steps performed when selection a new configuration. The component C sends to its upstream dependencies a new composite model in the following two cases:

- the reliability of the new configuration is higher than the reliability of the configuration configuration and the component C decides to switch to the new configuration;
- or the updated reliability of the current configuration is lower than the previous reliability.

In both cases, the changes in the reliability provided may trigger further architectural changes in higher levels of the composition hierarchy.

Although the same PCA model of each component is used in all configurations and restricted using the interface and hiding operators, the reliability provided by a component is not the same in different configurations as it is determined by the following factors:

- probability of failure actions in the PCA model of the component;
- how the component is used by upstream bindings;
- reliability provided by downstream dependencies.

For example, a component may have a failure action with a high probability but if that action is rarely used in a given configuration it will have a small impact on the overall reliability. While the probability associated with a transition is given by the component's execution profile, the frequency with which it is called depends on the probability of the output actions in upstream

components. Therefore, the reliability of a component is always defined with respect to a particular configuration. The descendent phase of the algorithm ensures that each component knows how it is going to be used by upstream components and the hiding operator ensures only interface actions are used in the behavioural descriptions exchanged. In the ascendant phase, the composite representation computed contains the propagation and handling of failures from downstream dependencies. *As a result, in the ascendant phase, the selection of the most reliable configuration is based on the reliability of downstream dependencies combined with the execution profile of upstream bindings propagated during the descendent phase.* Although each component makes a local choice, the descendent phase of the reconfiguration algorithm ensures that a component has all the information about the choices performed by upstream dependencies which enables the component to select a configuration that maximises the reliability of the composition hierarchy.

5.2.4. Example WebSystem for Distributed Reconfiguration Algorithm

In the case of the Web system presented in the last Section, the described distributed algorithm starts a descendent phase from the component instance *Client* as this component does not have provided interfaces. The *Client* projects its behaviour representation A_{Client} to the actions corresponding to its interface *getWebPage* ($\mathcal{E}(\text{getWebPage})$) using hiding operator and then sends it to the *Server*. The *Server* first receives the reduced PCA from *Client* which corresponds to the interactions of *Client* with *Server* (Algorithm 5.1 - line 1). The *Server* then uses the interface operator to remove the behaviour associated with unbound provided interfaces and the hiding operator to remove internal behaviour (Algorithm 5.1 - line 2). Thereafter, the *Server* computes the sub-set of its required interfaces of which its bound provided interfaces to *Client* are functionally dependent on (Algorithm 5.1 - lines 4 - 7). The set of *active* required interfaces is then used in combination with the available set of component instances (*Backend-1*) to generate all the possible bindings configurations to the active required interfaces of *Server* (Algorithm 5.1 - line 12): $\mathcal{B}_1 = \text{Backend-1.getImage} \mapsto \text{Server.getImage},$
 $\text{Backend-1.getContent} \mapsto \text{Server.getContent};$

For each configuration $\mathcal{B}$, the component instance *Server* computes the representation of each component instance C_i in $\mathcal{B}$ that corresponds to bindings between *Server* and C_i in $\mathcal{B}$ (Algorithm 5.1 - lines 13 - 15). This representation reflects how the *Server* uses each component instance in a given configuration $\mathcal{B}$ based on how its provided interfaces are used by an upper level component,

in this case *Client*, without including the behaviour associated with the interactions with the upper level component. The *Server* then waits for these component instances (*Backend-1* in this case) to send a PCA representation that reflects their downstream choices based on the same steps performed by *Server* (Algorithm 5.1 - line 17). The algorithm stops when the first phase reaches the leaf components *DB1* and *DB2*, *i.e.* without required interfaces, as no permutations of bindings to required interfaces can be calculated (Algorithm 5.1 - line 12). The distributed reconfiguration algorithm starts thereafter an ascendant phase where *DB1* and *DB2* send to *Backend-1* their interface PCA representation using the hiding operator (Algorithm 5.1 - line 25 - 28). Upon receiving the PCA representations from *DB1* and *DB2*, *Backend-1* calculates a composite PCA from these representations that denotes its bindings to *DB1* and *DB2*, reduces it using the hiding operator and sends it to the *Server*. The *Server* then repeats the same steps and sends its corresponding reduced PCA representation to the *Client*. The ascendant phase of the algorithm finishes when the *Client* receives the representation from *Server*, as the *Client* component does not have provided interfaces (Algorithm 5.1 - line 25). Although no central choice over alternative configurations to required interfaces is performed, each component selects a configuration based on the choices made by the *Client* and propagated along the composition hierarchy.

When *Backend-2* becomes available, the *Server* can consider the following alternative configurations to its required interfaces (algorithm 5.1 - line 11):

$$\begin{aligned}
 Server_{\mathcal{B}_1} = & \text{Backend-1.getImage} \text{ -- } Server.getImage, \\
 & \text{Backend-1.getContent} \text{ -- } Server.getContent \\
 Server_{\mathcal{B}_2} = & \text{Backend-2.getImage} \text{ -- } Server.getImage \\
 & \text{Backend-1.getContent} \text{ -- } Server.getContent
 \end{aligned}$$

In the next paragraphs we describe the steps performed by the distributed reconfiguration algorithm to compute the representations for configuration $\mathcal{B}_2$ (shown in Figure 5.6) as the representations for configuration $\mathcal{B}_1$ have been created in the initial run of the algorithm.

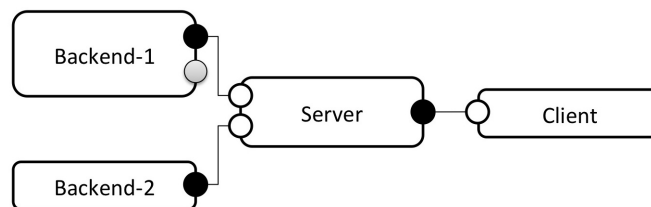
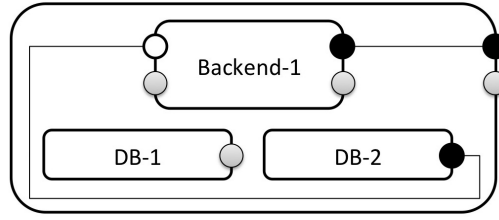


Figure 5.6.: Configuration Used to Generate Intermediate PCA for Server Component

In the process of computing the reliability of configuration $\mathcal{B}_2$, the *Server* uses the hiding operator to compute PCA representations that denote how the *Server* uses *Backend-1* and *Backend-2*, according to the bindings in $\mathcal{B}_2$. The *Server* then sends these representations to *Backend-1* and *Backend-2* and the distributed reconfiguration process continues from those components. After receiving the reduced PCA from *Server* (Algorithm 5.1 - line 1), *Backend-1* uses the interface operator to construct a PCA representation that includes the behaviour associated with the interface *getContent*, thereby removing the behaviour associated with its unbound interface *getImage*. Thereafter, *Backend-1* identifies which of its required interfaces need to be bound to provide the interface that is used by *Server* (Algorithm 5.1 - lines 4 - 7). For example, *Backend-1* identifies that *DB-2* is not needed to provide interface *getContent*, as shown in Figure 5.7. *Backend-1* then sends *DB-1* a reduced PCA representation which denotes how it uses the interface *getContent* and also how *Backend-1* is used by the *Server* (algorithm 5.1 - lines 15 and 16). As *DB-1* is a leaf component, it replies to *Backend-1* by sending its interface PCA representation (algorithm 5.1 - lines 27 and 28). When receiving the interface representation from *DB-1*, *Backend-1* computes the composite representation from the representations sent by its downstream dependencies ($DB - 1$) and its interface representation w.r.t. the *Server* (algorithm 5.1 - line 26). The steps performed by *Backend-2* are very similar to the ones performed by *Backend-1* and are therefore omitted.

The execution of the distributed assembly algorithm continues on the *Server* after the *Backend-1* and *Backend-2* have sent their representations to enable the *Server* to decide between the available alternative configurations (algorithm 5.1 - line 17). To this end, the *Server* calculates the composite representation corresponding to the configuration $\mathcal{B}_2$ using the representations sent by *Backend-1* and *Backend-2* and the PCA model of the *Server* (algorithm 5.1 - line 18). As this composite representation also includes how the *Client* uses the *Server*, it does not contain input actions and can therefore be translated to a DTMC to calculate the reliability of $\mathcal{B}_2$ provided to the *Client* component (algorithm 5.1 - line 20). In the case that the configuration $\mathcal{B}_2$ achieves a higher reliability than $\mathcal{B}_1$, the *Server* calculates a composite representations that includes its downstream dependencies (*Backend-1* and *Backend-2*) and sends it to the *Client* to enable it to make a decision on the new configuration. If the *Client* decides to change the reconfiguration of the system, then bindings defined by configuration $\mathcal{B}_2$ are established along the component hierarchy.

Figure 5.7.: Configuration of Backend-1 Component in System Configuration $\mathcal{B}_2$

5.3. Evaluation

Using the WebSystem example we evaluate the time taken by the both versions of the reconfiguration algorithm to compute the necessary PCA representations for the selection of alternative configurations. Of particular interest is the overhead introduced by the *distributed* re-assembly algorithm since it computes intermediate representations at each descending and ascending step. We use the *centralised* algorithm as a baseline for comparison.

In the *centralised* algorithm selecting the most reliable configuration is based on the composite representation of all components in each configuration. Thus, we compare in Table 5.1 the time required to construct and analyse each configuration individually and the total time denotes the time required to compute all configurations sequentially.

The *distributed* algorithm recursively constructs the intermediate representations that enable local choice at each component with alternative configurations for its required interfaces. In contrast with the centralised case, the time to construct each configuration denotes the time to construct all the intermediate representations, for both the descendent and the ascendant phases until the *Server* can select the configuration with the highest reliability. Note however that some of the computations could be done concurrently as the components are distributed. Nevertheless, for comparison purposes, we place ourselves in the worst case scenario and compute sequentially the intermediate PCA representations for $\mathcal{B}_1$ and $\mathcal{B}_2$. We ignore the network delay for transmitting the intermediate representations as such delays are dependent on the deployment context and the size of the reduced models is small (results in Table 5.1).

	Centralised	Distributed
Configuration $\mathcal{B}_1$	59ms	89ms
Configuration $\mathcal{B}_2$	54ms	92ms
Total	113ms	223ms

Table 5.1.: Architectural Assembly Evaluation Results

Although the WebSystem is only a small example and further evaluation at larger scales is certainly needed, the figures herein presented suggest that the overheads of computing intermediate representations are not prohibitive and can be easily compensated if the components execute concurrently. This is due to the gains in size obtained by reducing the PCA of each component to its interface representation based on the algorithm described earlier in Chapter 4.

5.4. Summary

(Self-) adaptation of autonomous systems, whether distributed applications or services is driven by non-functional aspects that need to be automatically, rather than manually, associated with their software models. This requires a close integration between the architectural, behavioural, non-functional and management concerns. Architectural models that do not consider the behaviour representation of each component cannot determine the functional dependency between provided and required interfaces of a single component as well as the compatibility between component interactions. Consequently, the generated configurations can include incompatible components that lead the system to inconsistent states as well as components with bindings to required functionality that is not used by *active* behaviour. The latter is of special importance to pervasive and ubiquitous systems as it leads to unnecessary allocation of resources. On the other hand, architectural models which rely upon textual notations or external monitoring components for the non-functional aspects of component cannot accurately calculate the aggregated non-functional values of each configuration. Composability of models – deriving a composite model from the models of its parts – is also a key requirement for autonomous systems to be able to reason and adapt. Even if models of non-functional properties are combined with architectural representations, if these models are not composable then the adaptation processes must rely on user input and pre-defined scenarios of change.

In the framework described in this chapter, including the reconfiguration algorithms, we ensure a close correspondence between the elements of the different views. Provided and required interfaces of each component specified in the architectural view using Darwin are associated with input and output interfaces in the PCA representation of each component included the behaviour view. Management services keep both architectural and behavioural views updated at run-time by maintaining a repository of available component instances and monitoring the execution profile of each component to update the probabilities of transitions labelled with internal and output actions. Bindings between provided and required interfaces in the architectural view

correspond to synchronisation of input-output actions in the behavioural view. When generating alternative configurations of the system's architecture, the interface operator is used to adjust the PCA representation of each component to reflect the behaviour associated with bound provided interfaces. The same representation is used to determine which required interfaces are needed to satisfy functional requirements. Moreover, an underlying IA model is extracted from the modified PCA representation of each component to analyse the compatibility of components with bindings between their provided and required interfaces, which is then used to filter invalid configurations that would lead the system to inconsistent states. For each valid architectural configuration, the corresponding composite PCA model is automatically constructed from the modified PCA representations of each component, which reflect the bindings specified by the configuration. Reliability properties of each configuration are then automatically analysed to enable autonomous selection of the configuration that maximises the global reliability of the system.

Reducing the behaviour of each component before constructing the composite model in the case of the centralised reconfiguration, and before exchanging information with other components in the distributed version of the reconfiguration algorithm, improves the feasibility of model based adaptation using composable models. The reported execution times suggest the approach can be used in scenarios where reconfiguration needs to be decided under a second. Another important aspect of reducing the behaviour to interface models is that it allows the suppliers of different autonomous components such as hardware devices to exchange models that do not expose their underlying (and often proprietary implementations), which cannot be removed from the PCA representation of such components as it is necessary to monitor the execution profile of internal computations. However, whether applications are composed of distributed autonomous components or built as centralised systems is not only determined by performance considerations but also driven by the use of different suppliers or different hardware devices. So the methodology we propose remains applicable in many contexts.

A self-managing (sub-)system can be seen as a composition of autonomous entities, which include both functional and management components. While management components need to be aware of system models and adaptation capabilities, third-party bindings support provided by Darwin enables a separation between functional and management components. Self-management also requires encapsulation of the components' internal behaviour. This is needed for commercial as well as technical reasons: internal behaviour should not be disclosed. Our ability to reduce

a component's behaviour to its interface behaviour using the hiding and interface operators preserves encapsulation defined in the architectural model, thereby enabling the components to exchange these interface representations to allow each autonomous component to make its own local reconfiguration decisions, whilst maximising the global reliability properties by taking into account how upstream dependencies use its provided functionality. When architectural, behavioural and management models are integrated, when models are composable and can be reduced to their interfaces, it is possible for systems made of autonomous components to re-configure themselves to preserve global non-functional requirements.

6. Evaluation

In this Chapter we illustrate how Probabilistic Component Automata can be applied in practice using an example e-Banking system from Java EE [Ora], where it is employed to demonstrate how different Java-based component technologies can be used to build Service-Oriented systems. This example system provides a realistic application, of a small-moderate size and with several sources of failure, thus enabling a practical evaluation for PCA and the reduction algorithm presented in Chapters 3 and 4.

We start by describing the e-Banking system, its functionality and architecture. We show how PCA can model the probabilistic behaviour of each component by manually translating Java code to Probabilistic Finite State Processes (P-FSP) using a list of systematic rules and extracting the execution profile of each component from execution traces. We then combine the constructed PCA for each component with the system architecture to automatically generate the composite model of the entire system. The composite model is employed to analyse relevant operational and reliability properties. These can be used to verify Service Level Agreements as well as to trigger adaptation processes. Finally, we evaluate the time complexity gains obtained by the hiding operator on the analysis of such properties.

6.1. e-Banking Case Study System

The e-Banking system is structured as a three-tier architecture present in many web-based applications: application layer, business logic layer and data management (see Figure 6.1).

The application layer consists of two types of clients: an **Application client** which is used by administrators to manage customers and accounts, and a **Web client** that is used by customers to access account histories and perform transactions. The **Web client** is built using JavaServer Faces technology, whereas the **Application client** is implemented using Java Swing. Both clients access the information related to customers, accounts, and transactions via the following Enterprise Java Beans (EJBs).

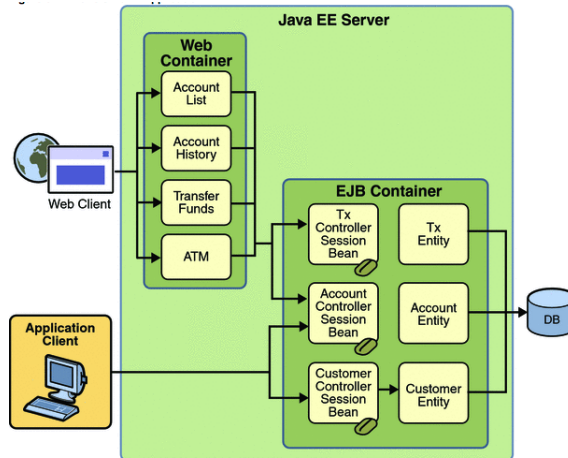


Figure 6.1.: Architectural Configuration of the e-Banking System from [Ora]

The **Transaction**, **Account** and **Customer** EJB components implement the business logic of the e-Banking system. These components are responsible for responding to client requests and performing the necessary updates on the Database. Note that not all the functionality provided by the EJB components is used by both clients. Access to the Database is conducted using Java Persistent Entities (JPEs) **Transaction**, **Account** and **Customer**, which provide an object view of the three tables in the Database with the same names. Each EJB component uses methods that encapsulate SQL queries from these JPEs to retrieve and update data from the Database.

6.1.1. Modelling Probabilistic Behaviour

In this Section we describe how we have manually extracted a P-FSP representation for each component in the e-Banking system from its Java source-code. These P-FSP are then compiled into PCA models which are independent from the context in which the components are deployed, and can hence be used in any configuration.

Each Java Persistent Entity (JPE) encompasses methods that encapsulate the queries performed on the Database. For example, Listing 6.1 shows an example of a query-method mapping. However, the code for the JPEs is not considered known, *i.e.* they are external components. As a result, their behaviour representation only contains the interface methods. The P-FSP representation of each method starts with an input action, which denotes that the method can be invoked, and is then followed by two possible outcomes: a) an output action that represents a successfully executed query and, when applicable, the result returned; b) an output failure

action denoting that an exception has been raised due to a failure during the query execution. The same mapping procedure can be applied in a general case where the implementation of external components is not known. In the case of the **Account Entity**, the translated P-FSP model for the code in Listing 6.1 and the corresponding PCA representation are depicted in Listing 6.2 and Figure 6.2, respectively.

Listing 6.1: Partial Java Code for AccountEntity JPE

```
@NamedQuery(name = "Account.findById",
    query = "SELECT a FROM Account a WHERE a.id = :id")
```

Although the main purpose of failure actions is to model failure scenarios, their semantics is flexible to accommodate other interpretation, as exceptions in the case of Java code.

Listing 6.2: Partial P-FSP for AccountEntity JPE

```
AccountEntity = (
    ?<1.0> accountFindByIDQuery ->
        ( !<0.99> accountFindByIDQueryResult -> AccountEntity
          | ~ !<0.01> accountFindByIDQueryResult -> ERROR
        )
    (...)
).
```

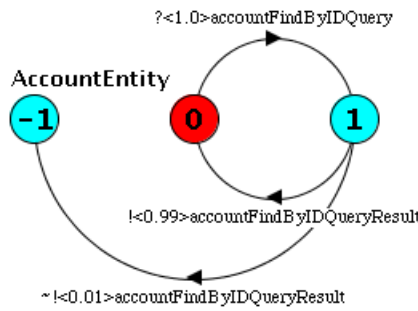


Figure 6.2.: Partial PCA Representation of Account Java Persistent Entity

We now illustrate the mapping between other Java constructs and P-FSP using the method `createAccount` in the `AccountControllerBean` (Listing 6.3).

Listing 6.3: Java Code for createAccount method of AccountControllerBeanJavaCode

```
public Long createAccount(AccountDetails details, Long customerId)
    throws IllegalAccountTypeException, CustomerNotFoundException,
        InvalidParameterException {
```

```
// makes a new account and enters it into db,
Account account = null;
Customer customer = null;

Debug.print("AccountControllerBean.createAccount");

if (details.getType() == null) {
    throw new InvalidParameterException("null_type");
} else if (details.getDescription() == null) {
    throw new InvalidParameterException("null_description");
} else if (details.getBeginBalanceTimeStamp() == null) {
    throw new
        InvalidParameterException("null_beginBalanceTimeStamp");
} else if (customerId == null) {
    throw new InvalidParameterException("null_customerId");
}

try {
    customer = em.find(
        Customer.class,
        new Long(customerId));

    if (customer == null) {
        throw new CustomerNotFoundException();
    }
} catch (Exception ex) {
    throw new EJBException(ex);
}

try {
    account = new Account(
        details.getType(),
        details.getDescription(),
        details.getBalance(),
        details.getCreditLine(),
        details.getBeginBalance(),
        details.getBeginBalanceTimeStamp());

    em.persist(account);
    account.addCustomer(customer);
} catch (Exception ex) {
    throw new EJBException(ex);
}

return account.getId();
}
```

The method signature is represented as an input action in the P-FSP model in Listing 6.4, indicating that the method can be invoked by another component. The interface exceptions `InvalidParameterException`, `IllegalAccountTypeException`, `CustomerNotFoundException` are represented as output failure actions, as these can be handled by other components. Although the `AccountControllerBean` handles the exceptions/failures related to interactions with the database, it raises an `ejbException` to the next level in the component hierarchy, *i.e.* to be handled by the components that interact with the `AccountControllerBean`. The interaction with the DB is performed by invoking the method `em.find (Customer.class , new Long(customerId))` , which is modelled in P-FSP using an output action corresponding to the invocation of the method `customerFindByIDQuery` offered by the `Customer` Entity. This is then followed by the input action `customerFindByIDQueryResult` to receive the query result, and the input failure action `customerFindByIDQueryResult`, which represents the failure handling behaviour when the `customerFindByIDQuery` fails.

Listing 6.4: P-FSP for createAccount method of AccountControllerBean

```

AccountControllerBean = (
  ?<1.0> createAccount -> <1.0> verifyParametersAccount ->
    (~!<0.01> invalidParameterExceptionAccountControllerBean ->
      ERROR
    |!<0.99> customerFindByIDQuery ->
      (~?<1.0> customerFindByIDQueryResult ->
        !<1.0> ejbExceptionAccountControllerBean -> ERROR
      |?<1.0> customerFindByIDQueryResult ->
        (!<0.01> customerNotFoundExceptionAccountControllerBean
          -> ERROR
        |!<0.99> accountCreateAccount ->
          (~?<1.0> accountCreateAccountResult ->
            ~!<1.0> ejbExceptionAccountControllerBean
              -> ERROR
          |?<1.0> accountCreateAccountResult ->
            !<1.0> executedCreateAccount
              -> AccountControllerBean
          )
        )
      )
    )
  )
).

```

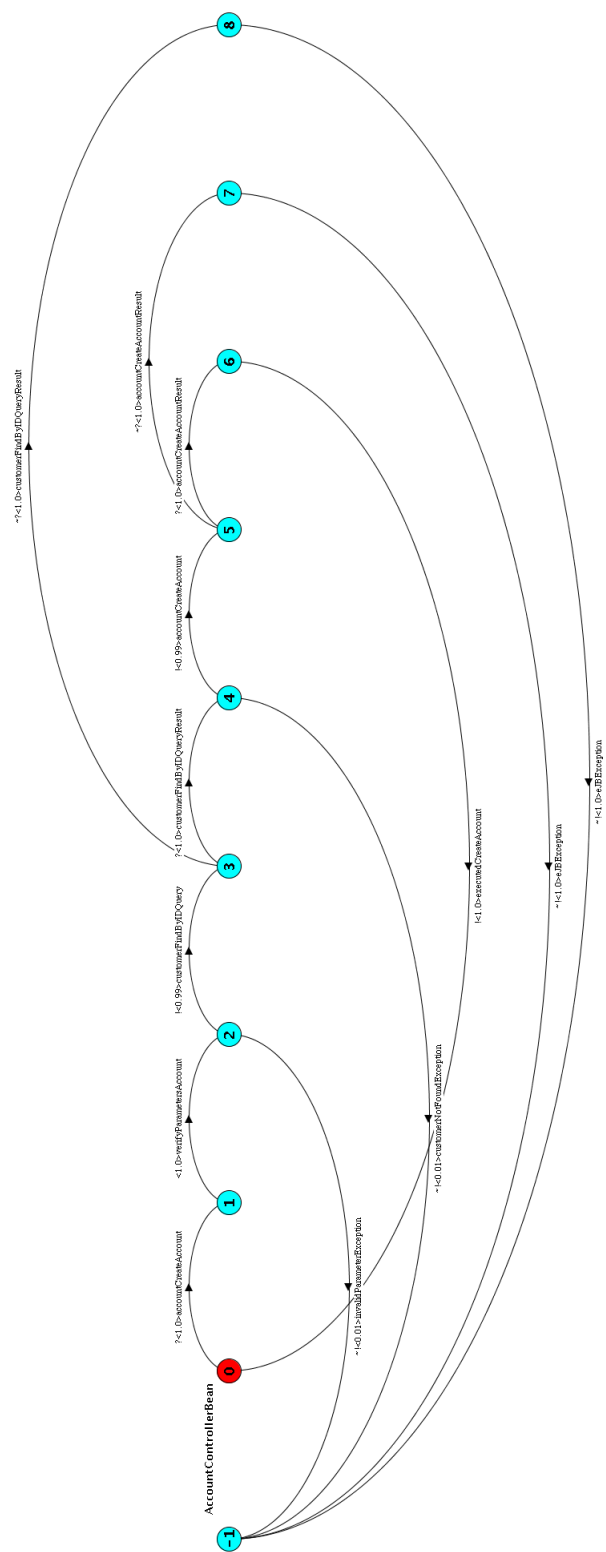


Figure 6.3.: PCA representation for createAccount method of AccountControllerBean

In general, the manual translation of an interface method in Java to P-FSP is based on the following rules:

- the method signature is denoted by an input action;
- internal actions, such as *if* statements and loops, are represented using internal actions;
- invocation of methods from other classes/components is depicted as output actions;
 - to model synchronous method calls, an output action is followed by an input action, even when the method does not have a return result;
 - if the method has declared some interface exceptions, these can be handled by using input failure actions with the same name.
- exceptions are represented as output failure actions;
- the final action of the interface method representation is an output action that denotes the return statement, which is implicit for methods without a return result.

Moreover, since different classes may have methods with the same name, the label of the input action that represents the method signature can be prefixed with the class name, e.g. *AccountControllerBean.createAccount*. When constructing the composite model of a system, this allows to distinguish the invocation of methods with the same name but from different classes, as synchronisation is performed based only on the action name.

In the next Section we cover how the P-FSP models translated from Java code are used to automatically construct the composite model of the entire e-Banking system.

6.1.2. Composite Model

The composite `AccountControllerBean` depicted in Figure 6.4 could be constructed using the parallel composition as follows:

`||CompositeAccountBean =`

`(AccountControllerBean || AccountEntity || CustomerEntity)`, where input/output actions associated with the bindings between provided/required interfaces are synchronised and internal actions are interleaved. However, not all the provided interfaces of the `AccountEntity` and `CustomerEntity` JPEs are used by the `AccountControllerBean`. We therefore remove the behaviour associated with unbound provided interfaces using the interface operator on the input

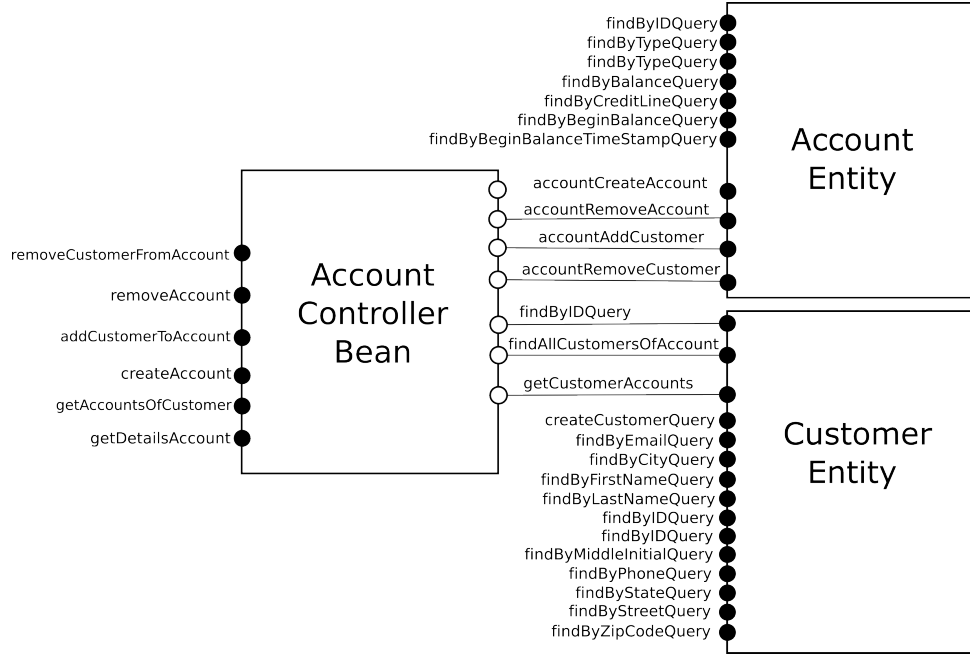


Figure 6.4.: Composite Account Controller Bean

actions that are not bound as composite representations with unused behaviour associated with unbound provided interfaces cannot be used for analysis of reliability and operational properties. For instance, the `AccountEntity` is constructed as: `AccountEntity = AccountEntityFull @ {findIDQuery, findByTypeQuery, findByBalanceQuery, findByCreditLineQuery, findByBeginBalance, findByBeginBalanceTimeStampQuery}`. The behaviour associated with these provided interfaces is kept as in the original representations, including all internal actions. All results reported hereafter are for models where unbound behaviour has been removed.

We construct the models for the `Customer Controller Bean` and `Transaction Controller Bean` in a similar way as: `CompositeCustomerBean = (CustomerControllerBean || CustomerEntity)`, `CompositeTransactionBean = (TransactionControllerBean || TransactionEntity)`. Furthermore, hierarchical composition is supported by composing composite models. For instance, the P-FSP expressions for the `Application Client` and `Web Client` are `CompositeAppClient = (WebClient || CompositeAccountBean || CompositeTransactionBean)` and `CompositeWebClient = (WebClient || CompositeAccountBean || CompositeTransactionBean)`. The sizes of the models produced by the previous P-FSP expressions are reported in Table 6.1.

While the previous P-FSP representations for the `Web Client` and the `Application Client` correspond to a single-threaded component with single bindings to each provided interface, in

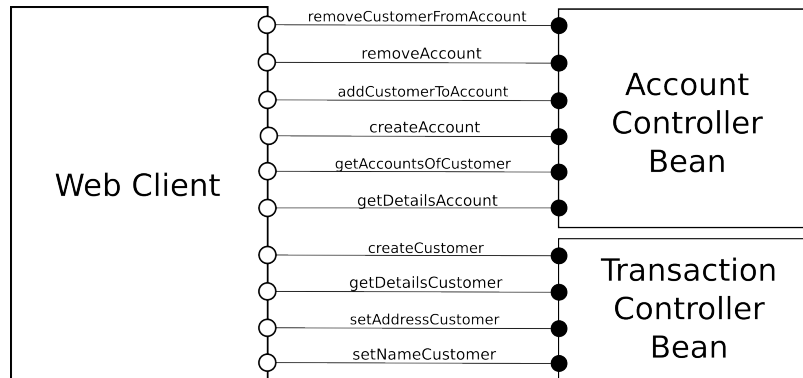


Figure 6.5.: Composite Web Client

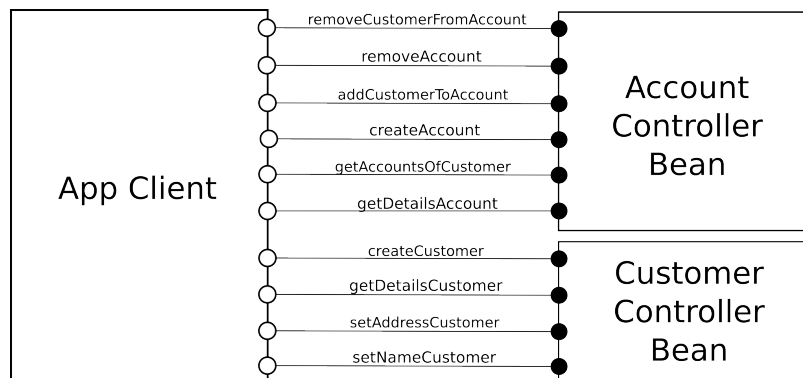


Figure 6.6.: Composite App Client

	#(States)	#(Transitions)
Account Controller Bean (ACB)	61	92
Account Entity	6	15
Customer Entity	3	6
Composite ACB	61	92
Customer Controller Bean (CCB)	35	52
Customer Entity	5	12
Composite CCB	35	52
Transaction Controller Bean (TCB)	50	90
Transaction Entity	3	6
Composite TCB	50	90
Application Client	38	75
Composite ACB	53	80
Composite CCB	27	40
Composite Application Client	116	161
Web Client	26	54
Composite ACB	11	15
Composite TCB	45	82
Composite Web Client	88	113

Table 6.1.: Size of PCA Models for Composition

the composite configuration of the e-Banking system the **Account Controller Bean** has multiple bindings to its provided interfaces as it used by both clients. Consequently, the interface actions of the **Account Controller Bean** need to be relabelled to allow for the representation of concurrent interactions with multiple clients. For example, the method `getDetailsAccount` in the **CompositeAccountBean** model is relabelled as follows:

```
||CompositeAccountBeanShared = CompositeAccountBean /
{ {getDetailsAccountWeb, getDetailsAccountApp} / getDetailsAccount,
{returnDetailsAccountWeb, returnDetailsAccountApp} / returnDetailsAccount }. Sim-
```

ilarly, the corresponding input/output actions in the **Web Client** and the **Application Client** also need to be relabelled to distinguish the requests from the two clients and to enable the synchronisation with the relabelled actions of the **Account Controller Bean**.

The final composite model for the entire e-Banking system has 8507 states and 24718 transitions and contains a large number of internal states and transitions. We have further applied the hiding operator to the components of the e-Banking system to reduce the model of

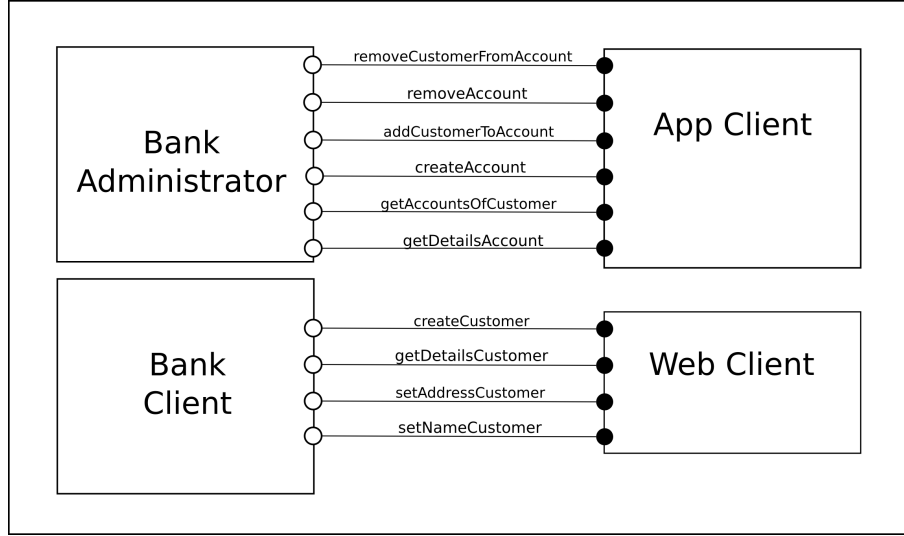


Figure 6.7.: Composite e-Banking System

each component to its interface behaviour before composition by removing internal behaviour. For example, the reduced model for the **AccountEntity** is computed as $||\text{AccountEntity} = \text{AccountEntityFull} \setminus \mathcal{E}_{\text{AccountEntityFull}}^{\text{int}}$, where *AccountEntityFull* represents the PCA model of the **Account Entity** without unbound provided interfaces. The gains obtained by reducing all the components of the system to their interface representation before composition are shown in Table 6.2. Since we had already removed behaviour associated with unbound interfaces, these figures represent primarily the removal of internal behaviour and the associated interleaved executions in the composite model. Even in this case the gains are significant and their implications in terms of analysis are discussed below.

	# States	# Transitions
Composite e-Banking Model	8507	24718
Reduced e-Banking Model	2246 (73.5 %)	8430 (65.9 %)

Table 6.2.: Size of Original and Reduced Composite Models for the e-Banking System

6.2. Probabilistic Analysis

To analyse the probabilistic characteristics of the e-Banking system, we need to associate probabilities with the output and internal actions of each component. These can be obtained by profiling components individually and counting the number of times output and internal actions have been executed. To this end, we have extended the source code of each component to log its execution profile. We avoided using a generic profiling tool to ensure the collected execution

traces reflect the mapping between Java code and P-FSP model defined in Section 6.1. Runtime traces can be collected on each component independently as a separate model is used for the representation of each component and these models are subsequently composed. When a sub-component needs to be replaced, only its execution profile needs to be discarded. *In contrast, modelling the entire system using a DTMC would require re-profiling the entire system whenever a sub-component needs to be replaced.*

Furthermore, we have emulated the execution of the **Application Client** and the **Web Client** to collect the traces of the calls on the other components. We then extracted the probabilities of transitions between states from the execution traces and analysed reliability properties based on the reachability of failure states. These properties can be used to verify if a system is complying with Service Level Agreements (SLAs) based on different levels of granularity. For example:

- reliability based the probability of the system failing;
- reliability experienced by a client component;
- probability of failure after requests of a client component;
- probability of failure of a specific component;
- probability of failure of a specific interface of a component.

	Description	PCTL Formula
R ₁	Probability of a system failure	$\mathcal{P}_{=?}[s = \text{fail} \cup \neg(s = \text{finish})]$
R ₂	Probability of a system failure after a request from the Application Client	$\mathcal{P}_{=?}[s = \text{fail} \wedge s = \text{appRequest} \cup \neg(s = \text{finish})]$
R ₃	Probability of a system failure after a request from the Web Client	$\mathcal{P}_{=?}[s = \text{fail} \wedge s = \text{webRequest} \cup \neg(s = \text{finish})]$
R ₄	Probability of a system failure caused by the Account Controller Bean	$\mathcal{P}_{=?}[s = \text{fail} \wedge s = \text{acbRequest} \cup \neg(s = \text{finish})]$
R ₅	Probability of a system failure caused by the <i>createAccount</i> of the Account Entity	$\mathcal{P}_{=?}[s = \text{fail} \wedge s = \text{createAccount} \cup \neg(s = \text{finish})]$
O ₁	Percentage of requests that are fully treated (without failure handling)	$\mathcal{P}_{=?}[s = \text{finish} \wedge \neg(\text{handled}) \cup \neg(s = \text{failure})]$

Table 6.3.: Reliability Properties for the e-Banking System

In table 6.3 we describe some specific cases for these properties in the context of the e-Banking system. For instance, the aggregated system's reliability measured by property R₁ is dependent on the components' reliability and their execution profile. Hence, each component may have

a different impact on the overall reliability of the system. Properties R_2 and R_3 are used to distinguish between the reliability experienced by each client, as the **Application Client** may require a higher reliability. Property R_4 determines the failures caused by the **Account Controller Bean** and the ratio between the values of R_4 and R_1 determines the impact of the **Account Controller Bean** in the overall system reliability. In general, this ratio can be used to select the components that should be re-implemented or replaced to improve overall reliability. Moreover, the source of the failures caused by a given component may not affect all the functionality provided by the component. With property R_5 we analyse the probability of failure caused by the interface *createAccount* of the **Account Controller Bean**. We include in table 6.3 an operational property O_1 which requires more actions to be kept in the model for analysis as all internal actions resulting from failure handling need to be kept.

We have analysed these properties in PRISM [KNP] for both the reduced and non-reduced versions of the composite representation. We automatically translate these composite models for the e-Banking system to DTMC models for analysis in PRISM. Note that such models need to be *closed*, *i.e.* without any input actions so that all transition probabilities are known. The resulting performance times for each of the properties considered are reported in table 6.4. The reported time includes both the time to analyse the property as well as the time to construct the composite model, and to reduce it in the case of the reduced model. We also include the size of the reduced model as it is specific to the actions needed for analysing each property (*e.g.* R_5).

	Analysis Time		Reduced Model Size	
	Full Model	Reduced Model	#(States)	#(Transitions)
R_1	715 s	46.89 s (93.44 %)	2246	8430
R_2	715 s	46.89 s (93.44 %)	2246	8430
R_3	715 s	46.89 s (93.44 %)	2246	8430
R_4	715 s	46.89 s (93.44 %)	2246	8430
R_5	715 s	47.29 s (93.38 %)	2310	8626
O_1	715 s	47.52 s (93.35 %)	2470	9116

Table 6.4.: Time to Analyse Reliability and Operational Properties for the e-Banking System

These results show large gains in the analysis times but their significance needs to be discussed more carefully. First, the results indicate that the gains obtained by applying the hiding operator before composing are much greater than the gains obtained by the state reduction techniques performed in PRISM on the overall DTMC model. This is because the size of the model is greatly reduced as shown in Table 6.2. **Second, we note that the average time to analyse both the full and the reduced models in PRISM is approximately the same for**

all properties as 99% of the reported time corresponds to the model building step. Once the model is internally constructed, checking each property requires an almost negligible time by comparison, *i.e.* approximately 100ms for the reduced model and 2s for the full model.

However, not all the properties can be analysed on the reduced model since some of the behaviour has been hidden. Thus, whilst the same model can be used for properties R_{1-4} which only require information about the components causing the failure, a different, finer granularity model is required to analyse R_5 . Hence, there is a tradeoff between the granularity of the properties that need to be checked and the time required for the analysis. This tradeoff is controlled by the user through the specification of the actions to be preserved/hidden by the reduction algorithm. **The significant gains in analysis time obtained suggest that often the user is better off re-building the model for the specific cases where fine granularity is required.**

Finally, this flexibility given to the user when analysing the reliability of the system can be achieved only when the models are composable, provide expressive failure representations and can be reduced. This is not the case in monolithic, non-composable models such as DTMCs, and not the case in many performance models that cannot be reduced through hiding.

6.2.1. Sensitivity Analysis

In this Section we perform sensitivity analysis on the e-Banking system to verify the impact of changes in the probability of failures for specific components. Component failures have different impacts on the system's reliability according to how the components are used. For instance, an unreliable component will not significantly affect the overall reliability if it is rarely used. On this account, we analyse the impact with respect to the execution profile of different users of the system, namely the Web Client, the Admin Client and both clients. The results obtained (Figure 6.8) show the impact on the system reliability for different probabilities of failure (1%, 2%, 5% and 10%) for the Tx Entity, Account Controller and Customer Controller components. The probability of failure for the other components was kept constant at 1% and we have specified that the Web Client generates 70% of the requests whilst the Application Client generates the rest 30%.

The sensitivity analysis for the Tx Entity shows that the Application Client is insensitive to the probability of the Tx Entity failing as it does not access information about the clients' trans-

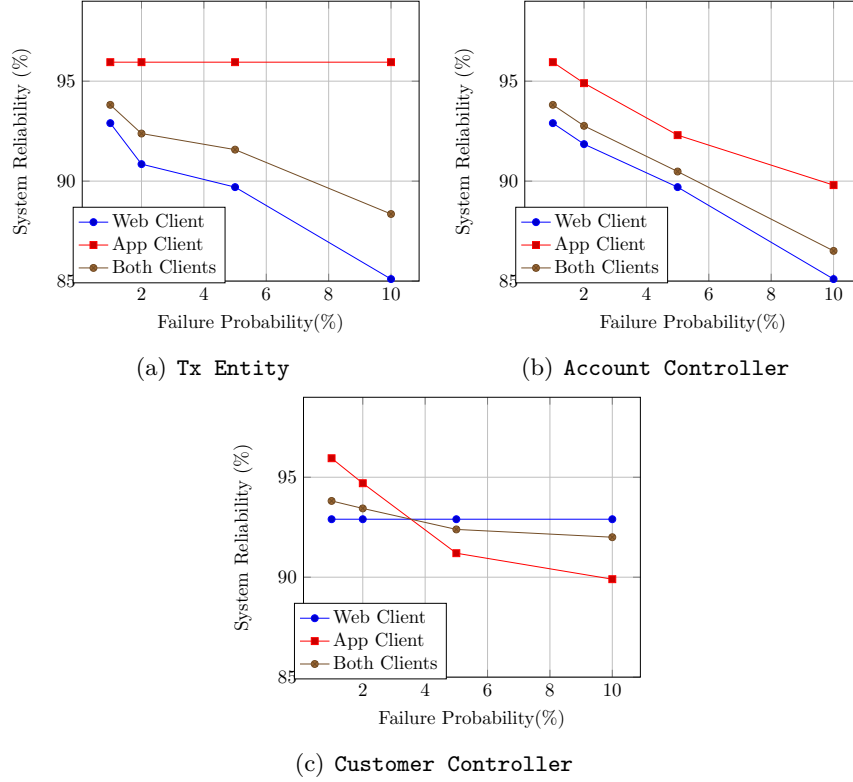


Figure 6.8.: Sensitivity Analysis on the e-Banking System

actions. Similarly, the Web Client is insensitive to the probability of the **Customer Controller** failing. As both clients interact with the **Account Controller** EJB, they are affected by changes in its failure probability.

Notwithstanding that the sensitivity analysis reported in Figure 6.8 is performed on the probability of failure of a given component, PCA models allow a system engineer to conduct fine-grained sensitivity analysis on the probability of failure of an individual action in a component. Although the **Account Controller** component is used by both the **Web Client** and **Application Client**, some of its interface actions are only available to the **Application Client**, *e.g.* `createAccount`. Figure 6.9(a) shows that the **Web Client** is insensitive to changes in the probability of failure of actions associated with the `createAccount` interface. *In contrast, if the probabilistic behaviour of the e-banking system is modelled as a DTMC following Cheung's approach [Che80], failure probabilities are associated only with specific components.* Consequently, the DTMC model does not include the behaviour associated with each provided interface, and more specifically the paths that lead the `AccountControllerBean` to fail. The results shown in Figure 6.9(b) illustrate the impact on the accuracy of reliability analysis.

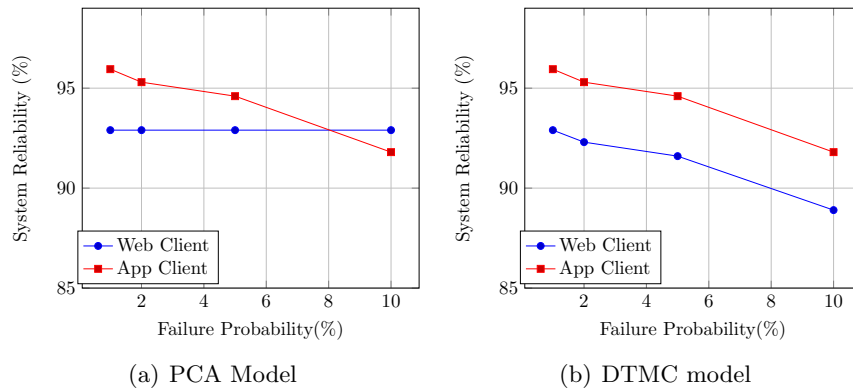


Figure 6.9.: Sensitivity Analysis on the `AccountControllerBean` Actions Using PCA and DTMC

6.3. Summary

We illustrated in this Chapter how P-FSP expressions can be used to model the behaviour of Java implementations of component-based applications. Invocation of methods is represented as output actions, while a method signature is denoted by an input action. The execution of internal instructions, such as if statements and loops are represented using internal actions. Failure actions provide a close mapping with exception modelling and handling in Java. Although the P-FSP expression for each component in the e-Banking system was manually defined and we have abstracted from many details such as variables, these expressions illustrate the flexibility and expressiveness of PCA models to model the probabilistic behaviour of component-based systems. The probabilities associated with transitions that are labelled with internal, output, internal and output failure actions can be automatically extracted from system profiling, thereby reducing the specification burden.

After translating P-FSP expressions into PCA models, the supported operators provide the means to adjust the generic representation of each component to one that is specific to a given system configuration. The interface operator enables automatic removal of behaviour associated with unbound provided interfaces. The re-labelling operator supports the representation of multiple bindings to a provided interface. The parallel composition allows automatic construction of the composite system model based on the (modified) representations of individual components. Therefore, a faithful composite representation of the system can only be obtained by establishing a close integration between the architectural and behavioural aspects of the system.

As the composite system model does not contain behaviour associated with unbound provided interfaces, it does not contain input actions and is automatically translated to a DTMC model.

We have illustrated how such model can be used to verify different reliability and operational properties. As the composite model includes the actions executed by the different components in the e-Banking system, it enables the definition of both global and granular properties. We have also shown the improvements in the accuracy of sensitivity analysis in comparison with a DTMC representation of the e-Banking following the modelling approach of existing work [Che80, FGT12], which abstracts from internal behaviour.

Furthermore, not all actions need to be kept in the composite model in order for some properties to be analysed. The PCA model of each component prior to composition by removing all internal actions that are not needed for analysis. We have shown that significant reductions in model size can be achieved. These, along with the gains obtained for the time to analyse each property, confirm the improvements in complexity illustrated in Chapter 4. The hiding operator allows the system designer to define the required actions to be preserved/hidden, thereby controlling the granularity of the composite representation for analysis.

6.4. Qualitative Evaluation

In this Section we discuss how the requirements defined in Chapter 1 have been met by the work described in this thesis.

Self-management systems need to be capable of changing autonomously, without user input at run-time. In this thesis we have focused on how the system architecture can be adapted to satisfy both functional and non-functional properties, with a focus on reliability. We have presented how the models of the system architecture, functional and probabilistic behaviour need to be closely integrated as considering such models in isolation leads to inconsistent analysis or requires user input. The multi-view framework we propose in Chapter 5 also integrates management elements for keeping models alive at run-time but also to determine the necessary reconfiguration steps to fulfil reliability requirements. Given that all elements of the multi-view framework are composable, the reconfiguration processes can consider any available configuration even if it was not defined at design-time. From the point-of-view of system analysis, the multi-view framework realises requirements R_1 (Autonomy), R_2 (Integrated System models) and R_3 (Composability). Moreover, the compositional reachability algorithm proposed in Chapter 4 presents an important contribution towards the use of composable models for self-management by improving the scalability of model construction and analysis (Requirement R_4).

The integration of models of functional behaviour and system architecture in the multi-view framework enables accurate analysis of the compatibility of components based on the bindings between provided and required interfaces, thereby ensuring adaptation safety before it is realised (Requirement R_5). We have also presented how the models in the multi-view framework can be exchanged between components to enable autonomous reconfiguration in a decentralised fashion (Requirement R_6).

Although we have presented contributions for autonomous analysis for reconfiguration, we have abstracted from lower-level details that need to be considered when fully integrating into a running autonomous system underpinned by a management framework. For example, we do not contemplate how components are allocated to nodes, as we assume that each component is instantiated in a separate node. The reconfiguration process needs to take into account the physical host where components are instantiated as it directly influences the non-functional properties achieved by each configuration. Other management aspects, such as security and resource usage, also have to be considered as they may prevent some configurations from being instantiated. The reconfiguration processes presented in this thesis take into account a single non-functional property: reliability. The addition of another non-functional property, such as performance, requires not only integration with existing models to avoid inconsistencies in the analysis but also requires modifying the reconfiguration process to a multi-objective framework.

7. Conclusions

The dynamic nature and increasing scale of modern pervasive systems and service-oriented environments demands the design of applications that are adaptive and autonomous. Consequently, manual maintenance performed by a system administrator is impractical and slow. Autonomous and self-managing systems require approaches that address non-functional properties, management and adaptation as first-class concerns.

Traditional Software Engineering models do not fully cater for these aspects which are often left for subsequent design iterations. On the other hand, adaptation approaches based on management services cater for the previous issues in isolation with the software model. We have discussed how these aspects should be combined to support self-management as they are clearly interdependent. We have also reviewed existing work on architectural modelling, behaviour analysis and model-based analysis of non-functional properties such as reliability and performance. Although some approaches integrate software models with adaptation processes, the lack of support for composability and compositionality prevents their systematic use. This limitation is more pronounced in the case of non-functional properties, which are needed to support adaptation choices.

We further reviewed in more detail different approaches to model probabilistic behaviour and discussed their limitations in the context of component-based systems and architectural adaptation. We described thereafter Probabilistic Component Automata (PCA), our formalism to model the probabilistic behaviour of those systems. By overcoming the main limitations of existing work, we proposed how the formalism can be closely integrated with architectural models. This enables the automatic construction of the composite model corresponding to a particular configuration of the system based on the individual representations of the system components. To mitigate the state-explosion problem associated with composite models, we introduced an algorithm that reduces the PCA representation of each component by removing its internal behaviour. We have then shown, both formally and empirically, that the reduced model preserves the properties of the original representation. Furthermore, using an example e-Banking appli-

cation from Java EE materials, we evaluate the expressiveness of PCA models to model the behaviour of component-based systems and demonstrate the gains in size reduction produced by reducing models before composition. Finally, we described how architectural, behavioural and non-functional aspects of a system are integrated in a multi-view modelling framework for self-management systems. This framework includes a management view which consists of management services that support architectural adaptation processes. We have defined a centralised and a distributed version which enable automatic selection of architectural configurations driven by global reliability requirements.

7.1. Contributions

The main contribution of this thesis is a composable modelling formalism, Probabilistic Component Automata, to represent the probabilistic behaviour of component-based systems. We have introduced constructs for failure modelling, propagation and handling whose semantics closely resemble the behaviour of exceptions in Object-Oriented languages. Building upon the existing features of the LTSA model checker, we have implemented tool support for PCA models by defining a probabilistic extension to FSP and extending the original operators to cater for the semantics of PCA models. All the existing tools for constructing, visualising and analysing composable models have also been extended to support PCA representations.

A close integration between PCA and architectural models of *provided* and *required* interfaces enables the definition of models that are independently from the context in which each component is deployed. By combining the semantics of generative and reactive probabilistic models, the representation of each component includes behaviour that a component controls and actions whose execution is chosen by others. In contrast with existing approaches based on DTMC models, the same PCA representation of each component can be used to construct the composite model of different configurations, though it may have to be adjusted before composition to reflect the bindings defined by each configuration. To this end, we have also defined interface and re-labelling operators which are used to remove behaviour associated with unbound provided interfaces and adjust the interface actions of components involved in multiple bindings to a provided interface, respectively. These operators allow us to adjust the PCA model of each component to reflect the bindings defined by each configuration. For instance, the interface operator removes the behaviour associated with unbound provided interfaces, thereby ensuring that the model used for composition does not contain input actions that are not going to be

synchronised with the corresponding output actions from other components. As a result, the composite model of a given architectural configuration does not contain input actions and can be translated to a DTMC model for reliability analysis. The reliability properties of each configuration are then automatically analysed using existing model-checking tools such as PRISM [KNP]. Our PCA formalism therefore provides the means to construct composite DTMC models for reliability analysis, in a similar way as PEPA models are used to construct the composite CTMC model of a system for performance analysis.

When analysing reliability properties of a system configuration, the same level of detail may not be required, in particular for system reconfiguration where components are replaced as a whole. The second main contribution of this thesis is the reduction algorithm presented in Chapter 4 which provides an efficient method to reduce a component's PCA representation to its interface actions by removing transitions labelled with internal actions. As a result, smaller composite models can be constructed by reducing the representations of sub-components w.r.t. to the reliability and operational properties to be analysed, before composing them using parallel composition. The reduction gains obtained can be significant, as demonstrated by the different examples used in this thesis. These results are supported by the lower time complexity of the algorithm (quasilinear), compared with existing methods for reducing generative probabilistic systems whose time complexity is cubic in the number of states of the original model. While existing methods are based on recursive processes that consider all the paths of the input model at each step, our reduction algorithm performs two iterative navigations whereby only local transitions to a single state are analysed at each step. Moreover, by formally showing the correctness of the algorithm using notions of behaviour equivalences, we have demonstrated that reduced models preserve the properties of the original properties and can be used to replace the original representation in any context (formula) defined by a combination of PCA operators. In particular to parallel composition, we are able to obtain significantly smaller composite representations by removing internal behaviour prior to composition, whilst preserving the reliability properties of a composite model with non-reduced representations.

We have leveraged existing work on architecture models and behaviour analysis and combined with PCA models to define a multi-view modelling framework for adaptive component-based applications. A close integration between the architectural, behavioural and non-functional aspects of a system underpins the definition of a centralised and a distributed algorithms for architectural reconfiguration driven by reliability requirements. Firstly, alternative configura-

tions are generated using the functional requirements established by the architectural model. Secondly, the components' PCA representations are used to determine which required functionality is needed for each component, based on the bindings to its provided interfaces. An underlying Interface Automata representation extracted from each PCA model is used to filter the configurations that would lead the system to inconsistent states. Thirdly, the reliability of each alternative configuration is calculated by composing the PCA representations of each component, which are first modified according to the bindings configuration. These constitute the main steps performed by both algorithms. In the case of the distributed version, although the choice between alternative configurations is performed on a local level by each component, the exchanged representations allow to preserve the global reliability of the system. Furthermore, as composability is preserved across all views of the modelling framework, reconfiguration processes do not require that all the architectural configurations considered at runtime to be defined at design time.

The use of component models that require explicit specification of the provided and required functionality of each component provides the means for third-party bindings determined by an external management component. Consequently, a clear separation between the functional and management behaviour can be established. Nonetheless, an autonomous component requires a close link between management elements and the models of the managed system.

In the next Section we describe several aspects in which the work presented in this thesis could be extended in the future.

7.2. Future Work

In the same way that probabilities of transitions labelled with internal, output, internal-failure and output failure actions are automatically extracted from run-time traces, the behaviour representation of a component should ideally be automatically derived from the source code. Although the PCA representations components of the example e-Banking system were manually defined, we followed some systematic rules. The same rules may be used as the basis for an extraction method that covers the main constructs of a particular language. Additionally, transitions in PCA models can be augmented with rewards to improve the expressiveness of properties that can be analysed using PCA representations, *e.g.* energy used. This extension requires the redefinition of the parallel composition, re-labelling and hiding operators based on the semantics of each reward structure.

Notwithstanding that the PCA representation of each component can be kept *live* at runtime, changes in the probabilities of transitions require the full reconstruction of the composite model for analysis of reliability properties. PCA models can be further extended to support late definition of the probability of some actions that are relevant for particular properties, *e.g.* interface and failure actions. The analysis of properties based on parametric models produces a formula that is dependent on the variables included in the composite model. As a result, reliability properties can be calculated later by directly replacing the updated probability of those actions. Similar to rewards, this extension requires the redefinition of the parallel composition, re-labelling and hiding operators.

Although we have employed PCA for reliability analysis of component-based systems to drive architectural reconfiguration processes, PCA is a formal and abstract probabilistic model whose semantics can be generalised and be applied in other contexts. For instance, Mallios *et al.* [MBK⁺13] extended PIOA for analysing the probabilistic cost of enforcing security policies. Additionally, PCA can also be further extended with other action types to accommodate alternative synchronisation semantics for other types of analysis.

The multi-view framework currently supports analysis of reliability properties, but other non-functional properties such as performance are equally important. Given the similarities between PEPA and Interface Automata, the performance behaviour of each component could be modelled using PEPA representations to support automated performance analysis. However, such representations would have to be integrated with existing views to avoid considering them in isolation. The similarities between single PEPA and Interface Automata models can provide a basis for such integration. For instance, *passive* active actions in PEPA are mapped to input actions in IA and PCA as the execution of these actions is determined by an external component. Although PEPA models do not distinguish between internal and output actions, as parallel composition requires the specification of set of actions for synchronisation, extending PEPA with output actions is straightforward. Finally, in order to support a full integration between software models, adaptation processes, management services and software components, the multi-view framework proposed in this thesis needs to be integrated with existing management frameworks that implement the management services required for architectural adaptation.

Bibliography

- [ACH98] Cristina Aurrecochea, Andrew T. Campbell, and Linda Hauw. A survey of qos architectures. *Multimedia Systems*, 6(3):138–151, 1998.
- [ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in archjava. In *Proc. of the 16th European Conf. on Object-Oriented Programming, ECOOP '02*, pages 334–367, 2002.
- [AG94] Robert Allen and David Garlan. Formalizing architectural connection. In *Proc. of the 16th Int. Conf. on Software engineering, ICSE '94*, pages 71–80, 1994.
- [AST09] Reza Asadollahi, Mazeiar Salehie, and Ladan Tahvildari. Starmx: A framework for developing self-managing java-based systems. In *Proc. of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 58–67, 2009.
- [BBS95] J. C. M. Baeten, J. A. Bergstra, and S. A. Smolka. Axiomatizing probabilistic processes: Acp with generative probabilities. *Inf. Comput.*, 121(2):234–255, September 1995.
- [BCL⁺06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257–1284, September 2006.
- [BCP⁺01] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: a Model Based Planner, 2001.
- [BH97] Christel Baier and Holger Hermanns. Weak bisimulation for fully probabilistic processes. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 119–130, 1997.

- [BHUB09] Cyril Ballagny, Nabil Hame Mocas: A state-based component model for self-adaptation. In *Proc. of the 2009 Third IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems, SASO '09*, pages 206–215, 2009.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [BKLS09] Nikola Bene, Jan K?etnsk, KimG. Larsen, and Ji? Srba. Checking thorough refinement on modal transition systems is exptime-complete. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, volume 5684 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin Heidelberg, 2009.
- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, April 1986.
- [CG12] Shang-Wen Cheng and David Garlan. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, December 2012.
- [Che80] R.C. Cheung. A user-oriented software reliability model. *Software Engineering, IEEE Transactions on*, SE-6(2):118 – 125, march 1980.
- [CK96] Shing Chi Cheung and Jeff Kramer. Context constraints for compositional reachability analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(4):334–377, October 1996.
- [CLG⁺09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors. *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, Berlin, Heidelberg, 2009.
- [CRMG08] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik. Early prediction of software component reliability. In *ICSE '08.*, pages 111 –120, may 2008.
- [CS99] Rance Cleaveland and Scott A. Smolka. Process algebra. *Information and Computation*, 87:58–77, 1999.
- [DA98] Luca De Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford, CA, USA, 1998. AAI9837082.

-
- [dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-9, pages 109–120, 2001.
- [DAO⁺11] Mahdi Derakhshanmanesh, Mehdi Amoui, Greg O’Grady, Jürgen Ebert, and Ladan Tahvildari. Graf: graph-based runtime adaptation framework. In *Proc. of the 6th Int. Symp. on Software engineering for adaptive and self-managing systems*, SEAMS ’11, pages 128–137, 2011.
- [DHK99] Pedro R. D’Argenio, Holger Hermanns, and Joost-Pieter Katoen. On generative parallel composition, 1999.
- [DKL⁺13] Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, Falak Sher, and Andrzej Wasowski. Abstract probabilistic automata. *Inf. Comput.*, 232:66–116, 2013.
- [DL06] Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive Fractal components. In *5th Int. Symp. on Software Composition (SC’06)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, March 2006. Springer-Verlag.
- [EEM10] Ahmed Elkhodary, Naeem Esfahani, and Sam Malek. Fusion: a framework for engineering self-tuning self-adaptive software systems. In *Proc. of the eighteenth ACM SIGSOFT Int. Symp. on Foundations of software engineering*, FSE ’10, pages 7–16, 2010.
- [EGMT09] Ilenia Epifani, Carlo Ghezzi, Raffaella Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE*, pages 111–121, Washington, DC, USA, 2009. IEEE Computer Society.
- [ESP⁺07] George Edwards, Chiyong Seo, Daniel Popescu, Sam Malek, and Nenad Medvidovic. Self-* software architectures and component middleware in pervasive environments. In *Proc. of the 5th Int. workshop on Middleware for pervasive and ad-hoc computing: held at the ACM/IFIP/USENIX 8th Int. Middleware Conf.*, MPAC ’07, pages 25–30, 2007.

- [EY09] Kousha Etesami and Mihalis Yannakakis. Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM*, 56(1):1:1–1:66, February 2009.
- [FGGM10] Antonio Filieri, Carlo Ghezzi, Vincenzo Grassi, and Raffaella Mirandola. Reliability analysis of component-based systems with multiple failure modes. In *Proceedings of the 13th international conference on Component-Based Software Engineering, CBSE'10*, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.
- [FGT12] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Form. Asp. Comput.*, 24(2):163–186, March 2012.
- [FKN⁺92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–57, 1992.
- [GCH⁺04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, October 2004.
- [GMM13] Vincenzo Grassi, Moreno Marzolla, and Raffaella Mirandola. Qos-aware fully decentralized service assembly. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13*, pages 53–62, 2013.
- [GMW97] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *Proc. of the 1997 Conf. of the Centre for Advanced Studies on Collaborative research, CASCON '97*, pages 7–, 1997.
- [GMW00] David Garlan, Robert T. Monroe, and David Wile. Foundations of component-based systems. chapter Acme: architectural description of component-based systems, pages 47–67. 2000.
- [Gok07] Swapna S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *Dependable and Secure Computing, IEEE Transactions on*, 4(1):32–40, jan.-march 2007.

-
- [GPS10] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. Programming language support to context-aware adaptation: a case-study with erlang. In *Proc. of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 59–68, 2010.
- [GS91] Susanne Graf and Bernhard Steffen. Compositional minimization of finite state systems. In *In Proc. 2ND International Conference of Computer-Aided Verification*, pages 186–196, 1991.
- [Hil96] Jane Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996.
- [HKMU06] Dan Hirsch, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Modes for software architectures. In *of Lecture Notes in Computer Science*, pages 113–126. Springer, 2006.
- [IBM] IBM. An architectural blueprint for autonomic computing. June.
- [IN08] Anne Immonen and Eila Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software & Systems Modeling*, 7(1):49–65, 2008.
- [Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.
- [KC03] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, jan 2003.
- [KCES07] Vibhore Kumar, Brian F. Cooper, Greg Eisenhauer, and Karsten Schwan. imanage: policy-driven self-management for enterprise-scale systems. In *Proc. of the ACM/IFIP/USENIX 2007 Int. Conf. on Middleware*, Middleware '07, pages 287–307, 2007.
- [KEC⁺08] Ivo Krka, George Edwards, Leslie Cheung, Leana Golubchik, and Nenad Medvidovic. A comprehensive exploration of challenges in architecture-based reliability estimation. In *WADS*, pages 202–227, 2008.

- [KGM10] Ivo Krka, Leana Golubchik, and Nenad Medvidovic. Probabilistic automata for architecture-based reliability assessment. In *Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*, QUOVADIS '10, pages 17–24, 2010.
- [KM90] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293 –1306, nov 1990.
- [KM97] S. Krishnamurthy and A.P. Mathur. On the estimation of reliability of a software system using reliabilities of its components. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 146 –155, 2-5 1997.
- [KNP] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *CAV'11*, volume 6806 of *LNCS*, pages 585–591. Springer.
- [KNP07] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.
- [KW04] J.O. Kephart and W.E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proc. of the Fifth IEEE Int. Workshop on Policies for Distributed Systems and Networks*, pages 3–, 2004.
- [LDS⁺08] Emil Lupu, Naranker Dulay, Morris Sloman, Joseph S. Sventek, Steven Heeps, Stephen Strowes, Kevin P. Twidle, Sye Loong Keoh, and Alberto E. Schaeffer Filho. Amuse: autonomic management of ubiquitous e-health systems. *Concurrency and Computation: Practice and Experience*, 20(3):277–295, 2008.
- [LKA⁺95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *Software Engineering, IEEE Transactions on*, 21(4):336 –354, apr 1995.

-
- [LMST10] Ruggero Lanotte, Andrea Maggiolo-Schettini, and Angelo Troina. Weak bisimulation for probabilistic timed automata. *Theoretical Computer Science*, 411(50):4291–4322, 2010.
- [LSW95] Kim Guldstrand Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In *Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '95, pages 17–40, London, UK, UK, 1995. Springer-Verlag.
- [LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151, 1987.
- [LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.
- [MBK⁺13] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, Fabio Martinelli, and Charles Morisset. Probabilistic cost enforcement of security policies. In Rafael Accorsi and Silvio Ranise, editors, *Security and Trust Management*, volume 8203 of *Lecture Notes in Computer Science*, pages 144–159. Springer Berlin Heidelberg, 2013.
- [MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *ESEC*, London, UK, 1995. Springer-Verlag.
- [Med96] Nenad Medvidovic. Formal modeling of software architectures at multiple levels of abstraction. In *In Proc. of the California Software Symp.*, pages 28–40, 1996.
- [MG05] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. 18th Int. Conf. on Architecture of Computing Systems*, 2005.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [MK06] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

- [MLD⁺11a] Marco Mori, Fei Li, Christoph Dorn, Paola Inverardi, and Schahram Dustdar. Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems. In *Proc. 9th Int. Conf. on Software engineering and formal methods*, SEFM'11, pages 286–301, 2011.
- [MLD⁺11b] Marco Mori, Fei Li, Christoph Dorn, Paola Inverardi, and Schahram Dustdar. Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems. In *SEFM*, pages 286–301, 2011.
- [MT10] Nenad Medvidovic and Richard N. Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 471–472, 2010.
- [OB99] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of guaraná. In *Proc 5th USENIX Conf. on Object-Oriented Technologies & Systems - Volume 5*, pages 15–15, 1999.
- [OMT08] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th Int. Conf. on Software engineering*, ICSE Companion '08, pages 899–910, 2008.
- [Ora] Oracle. Java ee 5 tutorial - bank application. <http://docs.oracle.com/javaee/5/tutorial/doc/bncma.html>.
- [PBU] Esteban Pavese, Víctor Braberman, and Sebastian Uchitel. Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models.
- [RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proc. of the 16th European Conf. on Object-Oriented Programming*, ECOOP '02, pages 205–230, 2002.
- [RCGS10] Rahul Raheja, Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Improving architecture-based self-adaptation using preemption. In *Proc. of the First Int. Conf. on Self-organizing architectures*, SOAR'09, pages 21–37, 2010.
- [SDK⁺95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21:314–335, April 1995.

-
- [SHMK07] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed architectural change for autonomous systems. In *Proc. of the 2007 Conf. on Specification and verification of component-based systems: 6th Joint Meeting of the European Conf. on Software Engineering and the ACM SIGSOFT Symp. on the Foundations of Software Engineering*, SAVCBS '07, pages 15–21, 2007.
- [SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From goals to components: a combined approach to self-management. In *Proc. of the 2008 Int. workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, pages 1–8, 2008.
- [SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *Proc. of the 2010 ACM Symp. on Applied Computing*, SAC '10, pages 431–438, 2010.
- [SL10] Morris Sloman and Emil C. Lupu. Engineering policy-based ubiquitous systems. *Comput. J.*, 53(7):1113–1127, 2010.
- [Slo] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2:333–360.
- [SM04] S.M. Sadjadi and P.K. McKinley. Act: an adaptive corba template to support unanticipated adaptation. In *Proc. 24th Int. Conf. on Distributed Computing Systems*, pages 4 – 83, 2004.
- [SMCS04] S. M. Sadjadi, P. K. McKinley, B. H. C. Cheng, and R. E. K. Stirewalt. TRAP/J: Transparent generation of adaptable java programs. In *Proc. of the 2004 International Symposium on Distributed Objects and Applications*, 2004.
- [SMK11] Daniel Sykes, Jeff Magee, and Jeff Kramer. Flashmob: Distributed adaptive self-assembly. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 100–109, New York, NY, USA, 2011. ACM.
- [SP99] Eugene W. Stark and Giridhar Pemmasani. Implementation of a compositional performance analysis algorithm for probabilistic i/o automata. In *In Proceedings of*

- 1999 Workshop on Process Algebra and Performance Modelling (PAPM99)*, pages 3–24, 1999.
- [ST07] N. Sato and K.S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In Bernd J. Krmer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing ? IC-SOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 107–118. Springer Berlin Heidelberg, 2007.
- [ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, May 2009.
- [SV04] A. Sokolova and E. P. De Vink. Probabilistic automata: System types, parallel composition and comparison. In *In Validation of Stochastic Systems: A Guide to Current Research*, pages 1–43. Springer, 2004.
- [Tin07] Simone Tini. Notes on generative probabilistic bisimulation. *Electron. Notes Theor. Comput. Sci.*, 175(1):77–88, May 2007.
- [TK93] Kuo-Chung Tai and Pramod V. Koppol. An incremental approach to reachability analysis of distributed programs. In *Proceedings of the 7th international workshop on Software specification and design, IWSSD '93*, pages 141–150, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [TK04] Gerald Tesauro and Jeffrey O. Kephart. Utility functions in autonomic systems. In *Proc. 1st Int. Conf. on Autonomic Computing*, pages 70–77, 2004.
- [VEBD07] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D’Hondt. Tranquillity: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *Software Engineering, IEEE Transactions on*, 33(12):856–868, dec. 2007.
- [VOKK] Christopher Van, Eenoo Osama, Hylooz Khaled, and M. Khan. Addressing non-functional properties in software architecture using adl.
- [VSS95] R.J. Vanglabbeek, S.A. Smolka, and B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121(1):59 – 80, 1995.

- [WPC06] Wen-Li Wang, Dai Pan, and Mei-Hwa Chen. Architecture-based software reliability modeling. *Journal of Systems and Software*, 79(1):132 – 146, 2006.
- [WSS94] Sue-Hwey Wu, Scott A. Smolka, and Eugene W. Stark. Composition and behaviors of probabilistic i/o automata. In *Theoretical Computer Science*, pages 513–528, 1994.
- [YCA04] S. Yacoub, B. Cukic, and H.H. Ammar. A scenario-based reliability analysis approach for component-based software. *Reliability, IEEE Transactions on*, 53(4):465 – 480, dec. 2004.
- [YY91] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the symposium on Testing, analysis, and verification*, TAV4, pages 49–59, New York, NY, USA, 1991. ACM.
- [ZCYM] Ji Zhang, Betty Cheng, Zhenxiao Yang, and Philip McKinley. Enabling safe dynamic component-based software adaptation. 3549:194–211.
- [ZZCD08] Fan Zhang, Xingshe Zhou, Junwen Chen, and Yunwei Dong. A novel model for component-based software reliability analysis. In *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, HASE '08, pages 303–309, Washington, DC, USA, 2008. IEEE Computer Society.

A. Notions of Equivalence for Probabilistic Component Automata

While in Chapter 4 we have discussed the reduction algorithm that implements the hiding operator, in this Chapter we show that the reduced model produced by the algorithm preserves the reachability properties of an input model. This corresponds to showing that the behaviour of the reduced model is equivalent to the one of the original model when abstracting from internal behaviour, which the algorithm tries to remove. In the context of automata based models, *strong* and *weak* bisimulation are the most used behaviour equivalences. These notions of behavioural equivalence can be informally described as follows. Consider that models A and B represent the interface behaviour of two components C_1 and C_2 where C_1 is being used in a certain system. If these models are strongly bisimilar, then component C_2 can replace C_1 as it can perform exactly the same behaviour as C_1 , *i.e.* A simulates the behaviour of B and vice versa. Weak bisimulation is similar to strong bisimulation but only considers the set of observable actions, *i.e.* two models A and B are weakly bisimilar if they can perform the same observable behaviour, excluding internal actions which are not externally observed. Consider that model A denotes a specification of a protocol and that model B represents an implementation of that protocol; if models A and B are weakly bisimilar, then the implementation is compliant with the specification of the protocol.

Formally, behavioural equivalences are defined based on a bisimulation relationship $\mathcal{R}$ that establishes a mapping between states which are considered equivalent, according to some properties [Hil96]. These properties are specific to both the behavioural equivalence and the model to which they are applied. Milner [Mil89] defined strong and weak bisimulations for non-probabilistic Labelled Transitions Systems, the corresponding bisimulation relationships $\mathcal{R}$ and associated properties. Moreover, Milner established that these bisimulation relationships $\mathcal{R}$ need to hold the following properties when applied to models A and/or B [Mil89]:

- reflexivity: $A \mathcal{R} A$;

- symmetry: $A \mathcal{R} B \iff B \mathcal{R} A$;
- transitive: $A \mathcal{R} B \wedge B \mathcal{R} C \implies A \mathcal{R} C$;
- congruence: $A \mathcal{R} B \implies C[A] \mathcal{R} C[B]$, for any context C .

An equivalence relationship $\mathcal{R}$ is a congruence if it is valid in any context, *i.e.* its properties are still valid after the application of any operator to models A and B .

In the next sections we will first review the notions of both strong and weak bisimulation for non-probabilistic and probabilistic versions of Labelled Transitions Systems. We then extend these notions to define the conditions for strong and weak bisimulation for PCA models. Thereafter, each case of the reduction algorithm described in the Chapter 4 is defined as a transformation T_i which is applied to an arbitrary state of an original model A . We demonstrate that each transformation T_i , when applied to a model A produces a weakly bisimilar model based upon a bisimulation relationship $\mathcal{R}_{T_i}$. We then show that a sequence of transformations T_i applied to model A also produces a weakly bisimilar model. This effectively corresponds to establishing that the bisimulation relationships $\mathcal{R}_{T_i}$ are transitive. Finally, we demonstrate that the bisimulation relationships $\mathcal{R}_{T_i}$ associated with each transformation T_i are a congruence with respect to the three operators supported by PCA: hiding, relabelling and parallel composition.

A.1. Behaviour Equivalences for Labelled Transition Systems

Consider a non-probabilistic LTS model $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta \rangle$, where:

- $\mathcal{S}$ is a set of states,
- q is the initial state,
- $\mathcal{E}$ is the set of labels and
- Δ is the set of transitions.

Although we are interested in using notions of strong and weak bisimulation between two LTS models, these notions are defined through a bisimulation $\mathcal{R}$ which defines an equivalence relationship between states of a single model. Consequently, in order for a bisimulation to be defined for two LTS models $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1 \rangle$ and $A_2 = \langle \mathcal{S}_2, q_2, \mathcal{E}_2, \Delta_2 \rangle$, a disjoint union $\hat{A} = \langle \hat{\mathcal{S}}, \hat{q}, \hat{\mathcal{E}}, \hat{\Delta} \rangle$ of automata A_1 and A_2 is defined as described by Lanotte *et al.* [LMST10]:

- $\hat{\mathcal{S}} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\hat{q}\} \mid \mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset \wedge \{\hat{q}\} \notin \mathcal{S}_1 \cup \mathcal{S}_2$;
- $\hat{\mathcal{E}} = \mathcal{E}_1 \cup \mathcal{E}_2$;
- $\hat{\Delta} = \Delta_1 \cup \Delta_2 \cup \{(\hat{q}, \tau, q_1) \cup (\hat{q}, \tau, q_2)\}$, where τ is a *silent* internal action.

In the following Sections we present the conditions for strong and weak bisimulations between two Labelled Transition Systems A_1 and A_2 based upon their disjoint union $\hat{A}$.

A.1.1. Strong Bisimulation

Milner [Mil89] defines a strong bisimulation between two LTS A_1 and A_2 through an equivalence relationship $\mathcal{R}$ on $\hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall a \in \hat{\mathcal{E}} \cup \{\tau\}$:

$$\text{if } s \xrightarrow{a} s' \implies \exists t' \in \hat{\mathcal{S}} \text{ such that } t \xrightarrow{a} t' \text{ and } (s', t') \in \mathcal{R}; \quad (\text{A.1})$$

$$\text{if } t \xrightarrow{a} t' \implies \exists s' \in \hat{\mathcal{S}} \text{ such that } s \xrightarrow{a} s' \text{ and } (s', t') \in \mathcal{R}. \quad (\text{A.2})$$

Note that there are many equivalence relationships $\mathcal{R}$ which verify the above conditions. For instance, consider the following two equivalence relationships:

- empty relationship $\mathcal{R} = \emptyset$;
- *reflexive* relationship, $\forall s \in \mathcal{S}_1, (s, s) \in \mathcal{R}$.

Although the above equivalence relationships are strong bisimulations as they verify the conditions A.1 and A.2, they are not sufficient to determine that two LTS models are strongly bisimilar. Two LTS A_1 and A_2 are strongly bisimilar, depicted by $A_1 \sim A_2$, if every step performed by A_1 is simulated by A_2 , and vice versa, in the same order [Mil89]. The two conditions for strong bisimulation ensure that if two states (s, t) are strongly bisimilar, then all destination states (s', t') of outgoing transitions from states s and t are also strongly bisimilar. The two conditions are also needed for $\mathcal{R}$ to be a symmetric relationship, *i.e.* it is not necessary to assume that if $(s, t) \in \mathcal{R}$ then $(t, s) \in \mathcal{R}$. However, the initial states (q_1, q_2) must also be mapped. It is therefore necessary and sufficient for two LTS models A_1 and A_2 to be strongly bisimilar iff there is an equivalence relationship $\mathcal{R}$ which is a strong bisimulation **and** $(q_1, q_2) \in \mathcal{R}$.

A.1.2. Weak Bisimulation

Weak bisimulation relaxes the conditions for strong bisimulation by abstracting from internal behaviour. In order to reflect this in the weak bisimulation definition, Milner introduced the

concept of a *weak* transition $s \xRightarrow{a} s'$ [Mil89] which corresponds to a sequence of zero or more non-observable transitions labelled with the internal action τ , followed by a transition labelled with action a and succeed by zero or more non-observable transitions labelled with the internal action τ : $s(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* s'$.

Weak bisimulation is defined through an equivalence relationship $\mathcal{R}$ on $\hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall a \in \hat{\mathcal{E}} \setminus \{\tau\}$:

- if $s \xrightarrow{a} s' \implies \exists t' \in \hat{\mathcal{S}}$ such that $t \xRightarrow{a} t'$ and $(s', t') \in \mathcal{R}$;
- if $t \xrightarrow{a} t' \implies \exists s' \in \hat{\mathcal{S}}$ such that $s \xRightarrow{a} s'$ and $(s', t') \in \mathcal{R}$.

Two models are weakly bisimilar if their observable behaviour is indistinguishable to an external observer, *i.e.* every visible step ($\xrightarrow{a}$) performed by an automaton A_1 is executed by A_2 ($\xRightarrow{a}$), though this step can be preceded and/or succeed by internal/silent actions τ which are not externally visible. Similarly to strong bisimulation, the existence of a weak bisimulation is not sufficient to determine that two models are weakly bisimilar. Formally, A_1 is weakly bisimilar to A_2 , denoted by $A_1 \approx A_2$, iff there is a weak bisimulation relationship $\mathcal{R}$ on $\hat{\mathcal{S}} \times \hat{\mathcal{S}}$ and $(q_1, q_2) \in \mathcal{R}$.

A.2. Behavioural Equivalences for Probabilistic Labelled Transitions Systems

Consider now a probabilistic LTS model (PLTS) $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$, an LTS model to which generative probabilities have been added to each transition. $\mu : \Delta \rightarrow [0, 1]$ is a function that assigns a probability to each transition in Δ subject to $\forall s \in \mathcal{S}, \sum_{(s, a, s') \in \Delta} \mu(s, a, s') = 1$, *i.e.* the sum of probabilities of all outgoing states from a given state s is equal to 1. In the following Sections we describe the notions of strong and weak bisimulation for PLTS models based on the work of Baier and Hermanns [BH97] using an alternative equivalent notation introduced by Tini [Tin07].

Let $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1, \mu_1 \rangle$ and $A_2 = \langle \mathcal{S}_2, q_2, \mathcal{E}_2, \Delta_2, \mu_2 \rangle$ be two PLTS models; similarly to the non-probabilistic case, an equivalence relationship $\mathcal{R}$ is defined on a disjoint union of the two automata $\hat{A} = \langle \hat{\mathcal{S}}, \hat{q}, \hat{\mathcal{E}}, \hat{\Delta}, \hat{\mu} \rangle$ [BH97], where:

- $\hat{\mathcal{S}} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\hat{q}\} \mid \mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset \wedge \hat{q} \notin \mathcal{S}_1 \cup \mathcal{S}_2$;
- $\hat{\mathcal{E}} = \mathcal{E}_1 \cup \mathcal{E}_2$;

- $\hat{\Delta} = \Delta_1 \cup \Delta_2 \cup \{ (\hat{q}, \tau, q_1) \cup (\hat{q}, \tau, q_2) \}$;
 - $\hat{\mu} : \hat{\Delta} \rightarrow [0, 1]$,
- $$\hat{\mu}(s, a, s') = \begin{cases} 0.5 & \text{if } (s, a, s') = (\hat{q}, \tau, q_1) \\ 0.5 & \text{if } (s, a, s') = (\hat{q}, \tau, q_2) \\ \mu(s, a, s') & \text{if } (s, a, s') \in \Delta_1 \\ \mu'(s, a, s') & \text{if } (s, a, s') \in \Delta_2 \end{cases}$$

The disjoint union PLTS $\hat{A}$ is equal to the one constructed for non-probabilistic LTS models, apart from the function $\hat{\mu}$. The first two cases of $\hat{\mu}$ ensure that the model $\hat{A}$ is a valid generative PLTS model, *i.e.* $\forall s \in \hat{\mathcal{S}}, \sum_{(s, a, s') \in \hat{\Delta}} \hat{\mu}(s, a, s') = 1$. This condition is valid for state $\hat{q}$ as well as for all other states given that A_1 and A_2 are valid PLTS models and their states are disjoint.

A.2.1. Strong Bisimulation

Two PLTS A_1 and A_2 are strongly bisimilar, depicted by $A_1 \sim A_2$, if every step performed by A_1 is simulated by A_2 , and vice versa, in the same order and with the same probability [BH97]. As a result, the strong bisimulation definition for PLTS models needs to consider not only the transitions of the two models but also their probability as defined by $\hat{\mu}$, which includes both μ_1 and μ_2 . Before describing Baier and Hermanns's strong bisimulation definition for PLTS models [BH97], we present some intermediate formulations to explain the changes required to the previous definitions of strong and weak bisimulation for LTS models, which arise from the introduction of probabilities.

We first consider a simple extension of the conditions for strong bisimulation for non-probabilistic LTS Models by adding μ to the previous rules. Given two PLTS Models A_1, A_2 and their disjoint union $\hat{A}$, we define a strong bisimulation $\mathcal{R}$ on $\hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall a \in \hat{\mathcal{E}} \cup \{\tau\}$:

$$\text{if } s \xrightarrow{a} s' \implies \exists t' \in \hat{\mathcal{S}} \text{ such that } t \xrightarrow{a} t', (s', t') \in \mathcal{R} \text{ and } \hat{\mu}(s, a, s') = \hat{\mu}(t, a, t'); \quad (\text{A.3})$$

$$\text{if } t \xrightarrow{a} t' \implies \exists s' \in \hat{\mathcal{S}} \text{ such that } s \xrightarrow{a} s', (s', t') \in \mathcal{R} \text{ and } \hat{\mu}(s, a, s') = \hat{\mu}(t, a, t'). \quad (\text{A.4})$$

If the previous conditions were applied to the models in Figure A.1, A_1 and A_2 would not be considered strongly bisimilar. However, model A_1 can perform a visible step with action a from state 0 to state 1 with probability $0.5 + 0.5 = 1$, while model A_2 can perform the same step with action a from state $0'$ to $1'$ with same probability 1. *Consequently, strong bisimulation for*

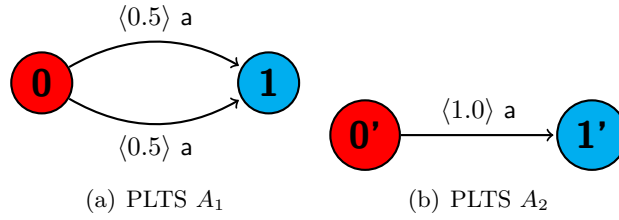


Figure A.1.: Examples of PLTS Models for Strong Bisimulation

PLTS models needs to be defined over sets of transitions between bisimilar states as opposed to single transitions.

Baier and Hermanns defined strong bisimulation for PLTS models [BH97] using a cumulative probability distribution function introduced by Vanglabbeek *et al.* [VSS95]. When applied to a PLTS $A = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$, the cumulative probability distribution function μ_G^1 determines the total probability by which a state $s' \in \mathcal{S}$ can be reached from state $s \in \mathcal{S}$ through transitions labelled with the action $a \in \mathcal{E}$:

$$\mu_G(s, a, s') = \sum_{(s, a, s') \in \Delta} \mu(s, a, s') \quad (\text{A.5})$$

$$\mu_G(s, a, s') = 0 \text{ iff } \nexists (s, a, s') \in \Delta \quad (\text{A.6})$$

Strong bisimulation for PLTS models is accordingly defined based on an equivalence relationship strong bisimulation $\mathcal{R} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall a \in \hat{\mathcal{E}} \cup \{\tau\}$ [BH97]:

- if $s \xrightarrow{a} s' \implies \exists t' \in \hat{\mathcal{S}}\{s'\}$ such that $t \xrightarrow{a} t', (s', t') \in \mathcal{R}$ and $\hat{\mu}_G(s, a, s') = \hat{\mu}_G(t, a, t')$;
- if $t \xrightarrow{a} t' \implies \exists s' \in \hat{\mathcal{S}}\{t'\}$ such that $s \xrightarrow{a} s', (s', t') \in \mathcal{R}$ and $\hat{\mu}_G(s, a, s') = \hat{\mu}_G(t, a, t')$.

Consequently, two states s and t are strongly bisimilar if the sum all outgoing transitions labelled with action a from state s to state s' is the sum as the sum all outgoing transitions labelled with action a from state t to state t' .

As the function μ_G already includes the existence of transitions between two states labelled with a certain action, Baier and Hermanns [BH97] defined the simplified conditions for strong bisimulation of PLTS models based on the notion of equivalent classes of states in $\hat{\mathcal{S}}$. Formally, the set of equivalence classes $\hat{\mathcal{S}}_C$ is determined by the binary relationship $\mathcal{R}$ as follows:

- $\hat{\mathcal{S}}_C = \hat{\mathcal{S}}/R,$

¹For convenience, we follow the notation used by Tini [Tin07], which is based on the definitions of Baier and Hermanns.

- $\hat{\mathcal{S}}_C = \{ [s]_{\mathcal{R}} \mid s \in \hat{\mathcal{S}} \}$, where
- $[s]_{\mathcal{R}} = \{ t \in \hat{\mathcal{S}} \mid (s, t) \in \mathcal{R} \}$

Moreover, before introducing the conditions for strong bisimulation [BH97], the function μ_G for a PLTS A needs to be extended to a set of target/destination states $\mathcal{S}_1 \subseteq \mathcal{S}$ as follows [Tin07]:

$$\mu_G(s, a, \mathcal{S}_1) = \sum_{s' \in \mathcal{S}_1} \mu_G(s, a, s'). \quad (\text{A.7})$$

A strong bisimulation for PLTS models is then defined by Baier and Hermanns [BH97] using an equivalence relationship $\mathcal{R} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}: \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \hat{\mathcal{E}} \cup \{\tau\}$:

$$\hat{\mu}_G(s, a, C) = \hat{\mu}_G(t, a, C). \quad (\text{A.8})$$

Analogously to the case of non-probabilistic LTS models, two PLTS models are strongly bisimilar iff there is a strong bisimulation $\mathcal{R}$ on $\hat{A}$ such that $(q_1, q_2) \in \mathcal{R}$. In the following paragraphs we will illustrate the application of bisimulation rules to two example PLTS models.

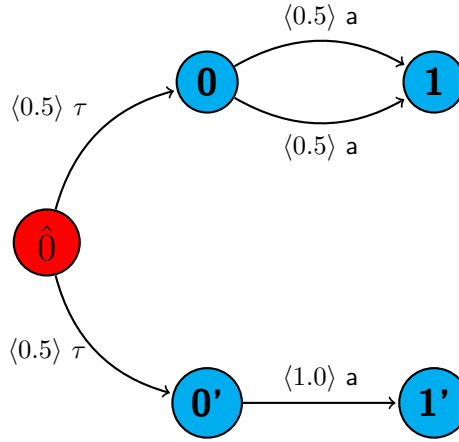


Figure A.2.: Disjoint Union PLTS $\hat{A}$ for Strong Bisimulation Between PLTS Models A_1 and A_2

Consider the PLTS $\hat{A}$ in Figure A.2 as the disjoint union of the PLTS models A_1 and A_2 from Figure A.1. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and $A_2 : \{(0, 0'), (1, 1')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{1, 1'\}\}^2$. Using the strong bisimulation definition in equation A.8, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a strong bisimulation.

²Note that for a bisimulation $\mathcal{R}'_{12} = \{ \{0, 0'\}, \{0, 1'\} \}$, the set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0', 1'\}\}$, *i.e.* there is only one equivalence class as states $0'$ and $1'$ are specified as being equivalent to state 0.

We start with the case where $(s, t) = (0, 0')$ and $C = \{0, 0'\}$ for the action label a :

$$\hat{\mu}_G(0, a, \{0, 0'\}) = \hat{\mu}_G(0', a, \{0, 0'\}). \quad (\text{A.9})$$

By applying the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$\hat{\mu}(0, a, 0) + \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) + \hat{\mu}(0', a, 0'). \quad (\text{A.10})$$

Given that the transitions $(0, a, 0)$, $(0, a, 0')$, $(0', a, 0)$ and $(0', a, 0')$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}(0, a, 0) = \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) = \hat{\mu}(0', a, 0') = 0, \quad (\text{A.11})$$

and we conclude that the condition for strong bisimulation is verified as follows for the current case

$$\hat{\mu}(0, a, 0) + \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) + \hat{\mu}(0', a, 0') \iff 0 + 0 = 0 + 0 \quad \square \quad (\text{A.12})$$

We omit the cases where $(s, t) = (1, 1')$ and $C = \{1, 1'\}$ as well as $(s, t) = (1, 1')$ and $C = \{0, 0'\}$ for the action label a as these are equivalent to the previous case since there are no transitions between the bisimilar states and the states in the equivalence classes.

Consider now the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the action label a :

$$\hat{\mu}_G(0, a, \{1, 1'\}) = \hat{\mu}_G(0', a, \{1, 1'\}). \quad (\text{A.13})$$

By applying the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$\hat{\mu}_G(0, a, 1) + \hat{\mu}_G(0, a, 1') = \hat{\mu}_G(0', a, 1) + \hat{\mu}_G(0', a, 1'). \quad (\text{A.14})$$

Given that the transitions $\hat{\mu}_G(0, a, 1')$ and $\hat{\mu}_G(0', a, 1)$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}_G(0, a, 1) + 0 = 0 + \hat{\mu}_G(0', a, 1'). \quad (\text{A.15})$$

By replacing $\hat{\mu}_G(0, a, 1)$ by $\mu_{1G}(0, a, 1)$ and $\hat{\mu}_G(0', a, 1')$ by $\mu_{2G}(0', a, 1')$ we confirm that the condition for strong bisimulation is also verified for this case:

$$0.5 + 0.5 = 1 \quad \square \tag{A.16}$$

As a result, the equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for strong bisimulation defined by equation A.8 and includes the initial states 0 and 0' of A_1 and A_2 , respectively. As a consequence, the PLTS models A_1 and A_2 are strongly bisimilar.

A.2.2. Weak Bisimulation

When establishing the notion of weak bisimulation for PLTS models, Baier and Hermanns [BH97] define the probability of *weak* transition $s \xRightarrow{a} s'$, *i.e.* the probability of paths from states s and s' which contain action a and may be preceded and/or followed by transitions labelled with internal action τ . To this end, consider the extension of the function μ_G for a PLTS A to sequences of actions in $\mathcal{E}^*$ [Tin07], *i.e.* with zero or more actions in $\mathcal{E}$. Let Λ be a sequence of actions in $\mathcal{E}^*$ and ϵ the empty sequence of actions. The function μ_G is defined as follows $\forall \Lambda \in \mathcal{E}^*$ and $\forall s \in \mathcal{S}$ and $\forall \mathcal{S}_1 \subset \mathcal{S}$:

$$\mu_G(s, \Lambda, \mathcal{S}_1) = \begin{cases} 1 & \text{if } \Lambda = \epsilon \wedge s \in \mathcal{S}_1 & \text{(A.17)} \\ 0 & \text{if } \Lambda = \epsilon \wedge s \notin \mathcal{S}_1 & \text{(A.18)} \\ \sum_{s' \in \mathcal{S}} \left(\mu_G(s, a, s') \cdot \mu_G(s', \lambda, \mathcal{S}_1) \right) & \text{if } \Lambda = a\lambda, a \in \mathcal{E} \wedge \lambda \in \mathcal{E}^* & \text{(A.19)} \end{cases}$$

Equation A.19 denotes an expansion law to calculate the probability over all the paths starting from state s to states in $\mathcal{S}_1$ through a sequence of transitions as defined by the sequence Λ . Consider the case where the function μ_G is applied to $(s_1, a, \{s_2\})$ and $(s_1, a, s_2) \in \Delta$. As the sequence of actions Λ in $\mathcal{E}^*$ is the label a followed by an empty sequence of actions ϵ ($\Lambda = a\epsilon$)³, we obtain:

$$\mu_G(s_1, a, \{s_2\}) = \sum_{s' \in \mathcal{S}} \left(\mu_G(s_1, a, s') \cdot \mu_G(s', \epsilon, \{s_2\}) \right). \tag{A.20}$$

As the summation in equation A.19 is defined over all states s' in $\mathcal{S}$, equation A.18 *removes* from the summation transitions labelled with action a from state s_1 to states s' which do not belong

³The case where $\Lambda = \epsilon a$ is equivalent.

to a path between state s_1 and states in $\mathcal{S}_1$, where $\mathcal{S}_1 = \{s_2\}$ in this case. In addition, equation A.17 denotes the identity case for $\mu_G(s', \epsilon, \{s_2\})$, $s' = s_2$ and ϵ denotes an empty sequence of actions.

Based on the extension of the function μ_G for a PLTS A to sequences of actions in $\mathcal{E}^*$, weak bisimulation for PLTS models is then defined through an equivalence relationship $\mathcal{R} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \hat{\mathcal{E}}$ [BH97]:

$$\hat{\mu}_G(s, \tau^* a \tau^*, C) = \hat{\mu}_G(t, \tau^* a \tau^*, C). \quad (\text{A.21})$$

$$\hat{\mu}_G(s, \tau^*, C) = \hat{\mu}_G(t, \tau^*, C). \quad (\text{A.22})$$

The first condition determines that two states s and t are weakly bisimilar if the sum of the probability over all the paths over a sequence of zero or more internal actions followed by a visible action a and a sequence of zero or more internal actions between them and states in the equivalence class C is the same. Note that the paths between state s and states in C may not have the same length as the corresponding paths between state t and C . On the other hand, the second condition ensures that two states s and t are weakly bisimilar if the sum of the probability over all the paths over a sequence of zero or more internal actions between them and states in the equivalence class C is the same. Formally, two PLTS models A_1 and A_2 are weakly bisimilar iff there is a weak bisimulation $\mathcal{R}$ on $\hat{A}$ such that $(q_1, q_2) \in \mathcal{R}$.

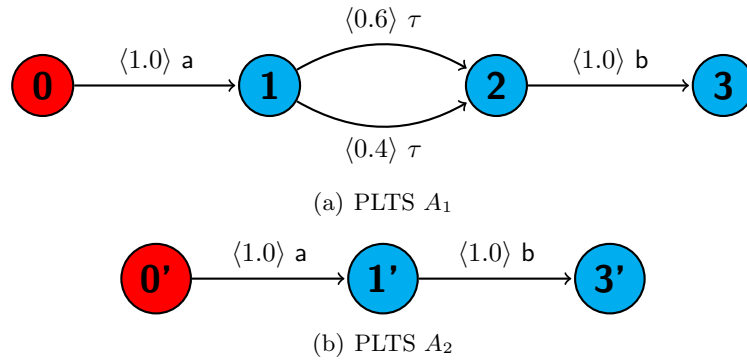


Figure A.3.: Examples of PLTS Models for Weak Bisimulation

Consider the example PLTS models A_1 and A_2 in Figure A.3 and their disjoint union $\hat{A}$ in Figure A.4. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and A_2 : $\{(0, 0'), (1, 1'), (3, 3')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{1, 1'\}, \{3, 3'\}\}$. Using the weak bisimulation definition in equation A.21, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a weak bisimulation.

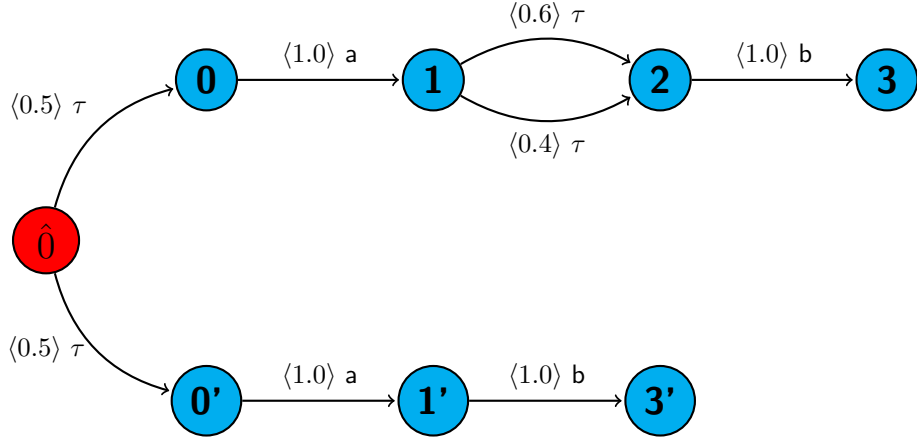


Figure A.4.: Disjoint Union PLTS $\hat{A}$ for Weak Bisimulation Between PLTS Models A_1 and A_2

We start by considering the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the action label a :

$$\hat{\mu}_G(0, \tau^* a \tau^*, \{1, 1'\}) = \hat{\mu}_G(0', \tau^* a \tau^*, \{1, 1'\}). \quad (\text{A.23})$$

Given that there is a single path between states 0 and 1 as well as between states $0'$ and $1'$ we derive the following from the expansion rule A.19:

$$\hat{\mu}_G(0, \epsilon, 0) \cdot \hat{\mu}_G(0, a, \{1, 1'\}) = \hat{\mu}_G(0', \epsilon, 0') \cdot \hat{\mu}_G(0', \tau^* a \tau^*, \{1, 1'\}). \quad (\text{A.24})$$

By applying rule A.17 to $\hat{\mu}_G(0, \epsilon, 0)$ and $\hat{\mu}_G(0', \epsilon, 0')$ as well as the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$1 \cdot (\hat{\mu}_G(0, a, 1) + \hat{\mu}_G(0, a, 1')) = 1 \cdot (\hat{\mu}_G(0', a, 1) + \hat{\mu}_G(0', a, 1')). \quad (\text{A.25})$$

Given that the transitions $\hat{\mu}_G(0, a, 1')$ and $\hat{\mu}_G(0', a, 1)$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}_G(0, a, 1) + 0 = 0 + \hat{\mu}_G(0', a, 1'). \quad (\text{A.26})$$

By replacing $\hat{\mu}_G(0, a, 1)$ by $\mu_{1G}(0, a, 1)$ and $\hat{\mu}_G(0', a, 1')$ by $\mu_{2G}(0', a, 1')$ we confirm that the condition for weak bisimulation is verified for this case:

$$1 = 1 \quad \square \quad (\text{A.27})$$

Consider now the case where $(s, t) = (1, 1')$ and $C = \{3, 3'\}$ for the action label b :

$$\hat{\mu}_G(1, \tau^* b \tau^*, \{3, 3'\}) = \hat{\mu}_G(1', \tau^* b \tau^*, \{3, 3'\}). \quad (\text{A.28})$$

While there is only one transition in the path between states $1'$ and $3'$ in $\hat{A}$, the paths between states 1 and 3 include transitions through intermediate state 2 . Therefore, the general condition A.21 for weak bisimulation is mapped to:

$$\hat{\mu}_G(1, \tau b, \{3, 3'\}) = \hat{\mu}_G(1', b, \{3, 3'\}). \quad (\text{A.29})$$

Using the expansion rule A.19 on $\hat{\mu}_G(1, \tau b, \{3, 3'\})$ we obtain:

$$\sum_{s \in \hat{S}} \hat{\mu}_G(1, \tau, s) \cdot \hat{\mu}_G(s, b, \{3, 3'\}) = \hat{\mu}_G(1', b, \{3, 3'\}). \quad (\text{A.30})$$

Given that all the transitions from state 1 only have state 2 as the destination state, then for all states $s \in \hat{S} \setminus \{2\}$, $\hat{\mu}_G(1, \tau, s) \cdot \hat{\mu}_G(s, b, \{3, 3'\}) = 0$ as $\hat{\mu}_G(1, \tau, s) = 0$. As only the transition between states 1 and 2 has a probability greater than 0 , we obtain the following by also applying the rule A.7 when μ_G is applied to a set of states:

$$\hat{\mu}_G(1, \tau, 2) \cdot (\hat{\mu}_G(2, b, 3) + \hat{\mu}_G(2, b, 3')) = \hat{\mu}_G(1', b, 3) + \hat{\mu}_G(1', b, 3'). \quad (\text{A.31})$$

By replacing $\hat{\mu}_G$ by the corresponding cumulative probability functions from A_1 and A_2 we derive

$$\mu_{1G}(1, \tau, 2) \cdot (\mu_{1G}(2, b, 3) + 0) = 0 + \mu_{2G}(1', b, 3'). \quad (\text{A.32})$$

and confirm that the condition for weak bisimulation is also verified for this case:

$$(0.5 + 0.5) \cdot 1 = 1 \quad \square \quad (\text{A.33})$$

The equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for weak bisimulation defined by equation A.21 and includes the initial states 0 and $0'$ of A_1 and A_2 , respectively. As a consequence, the PLTS models A_1 and A_2 are weakly bisimilar.

A.3. Behaviour Equivalences for Probabilistic Component Automata

After reviewing the definitions of strong and weak bisimulations for LTS [Mil89] and PLTS [BH97] models, in this Section we extend these notions of behaviour equivalence to define the conditions for strong and weak bisimulation for PCA models. We start by recalling the definition of a PCA model. A PCA is a tuple $\langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$, where

- $\mathcal{S}$ is a set of states;
- $q \in \mathcal{S}$ is the initial state;
- $\mathcal{E}$ is a set of action labels;
 - $\mathcal{E} = \mathcal{E}^{in} \cup \mathcal{E}^{loc} \cup \mathcal{E}^{fail}$;
 - $\mathcal{E}^{in}$ is the set of input action labels;
 - $\mathcal{E}^{loc} = \mathcal{E}^{int} \cup \mathcal{E}^{out}$ is the set of locally controlled action labels, where
 - * $\mathcal{E}^{out}$ is the set the of output action labels;
 - * $\mathcal{E}^{int}$ is the set of internal action labels;
 - $\mathcal{E}^{fail} = \mathcal{E}^{fail-in} \cup \mathcal{E}^{fail-out} \cup \mathcal{E}^{fail-int}$, where
 - * $\mathcal{E}^{fail-in}$ is the set of input failure actions;
 - * $\mathcal{E}^{fail-out}$ is the set of output failure actions;
 - * $\mathcal{E}^{fail-int}$ is the set of internal failure actions;
- $\Delta \subseteq \mathcal{S} \times \mathcal{E} \times \mathcal{S}$ is the set of transitions;
- $\mu : \Delta \rightarrow [0, 1]$ is a function that assigns a probability to each transition in Δ .

While in a PLTS model there is no explicit type for action labels and internal (non-observable) actions are denoted by a special label τ , in PCA internal actions are explicitly distinguished from visible actions (input and output). Therefore, internal behaviour is characterised by internal actions. Additionally, locally controlled actions (internal and output) follow the same proba-

bilistic semantics as in PLTS, *i.e.* the sum of the probabilities of all locally controlled actions from a given state s is 1:

$$\forall s \in \mathcal{S}, \left(\sum_{\substack{(s,a,s') \in \Delta \\ a \in \mathcal{E}^{loc}}} \mu(s, a, s') \right) = 1. \quad (\text{A.34})$$

On the other hand, input actions follow the semantics of reactive systems, *i.e.* the sum of probabilities of transitions labelled with a given input action from a given state is one, since the choice over input actions is performed by another PCA model:

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{E}^{in}, \left(\sum_{(s,a,s') \in \Delta} \mu(s, a, s') \right) = 1. \quad (\text{A.35})$$

Note that the previous definition of function μ_G for PLTS models can be used to calculate the probability of paths in PCA models without having to *resolve* the choices over input actions, as long as those paths do not include more than one input action. We describe how the choices over input actions are *resolved* when discussing weak bisimulation in relation to composite models. Although failure behaviour is modelled using separate failure actions, input failure actions follow the semantics of normal input actions, while output and internal failure actions follow the semantics of locally controlled actions. As a result, failure actions do not require special cases.

Similarly to LTS and PLTS models, the notions of bisimulation for PCA models are defined through an equivalence relationship $\mathcal{R}$ on a disjoint union of two models. Given two PCA models $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1, \mu_1 \rangle$ and $A_2 = \langle \mathcal{S}_2, q_2, \mathcal{E}_2, \Delta_2, \mu_2 \rangle$ we define their disjoint union PCA Model $\hat{A} = \langle \hat{\mathcal{S}}, \hat{q}, \hat{\mathcal{E}}, \hat{\Delta}, \hat{\mu} \rangle$, where:

- $\hat{\mathcal{S}} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\hat{q}\}$, such that $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$ and $\hat{q} \notin \mathcal{S}_1 \cup \mathcal{S}_2$;
- $\hat{\mathcal{E}} = \mathcal{E}_1 \cup \mathcal{E}_2 \cup \{\tau\}$;
- $\hat{\Delta} = \Delta_1 \cup \Delta_2 \cup \{(\hat{q}, \tau, q_1), (\hat{q}, \tau, q_2)\}$;

- $\hat{\mu} : \hat{\Delta} \rightarrow [0, 1]$

$$\forall (s, a, s') \in \hat{\Delta}, \hat{\mu}(s, a, s') = \begin{cases} 0.5 & \text{if}(s, a, s') = (\hat{q}, \tau, q_1) \\ 0.5 & \text{if}(s, a, s') = (\hat{q}, \tau, q_2) \\ \mu_1(s, a, s') & \text{if}(s, a, s') \in \Delta_1 \\ \mu_2(s, a, s') & \text{if}(s, a, s') \in \Delta_2 \end{cases}$$

A.3.1. Strong Bisimulation

Two PCA A_1 and A_2 are strongly bisimilar, depicted by $A_1 \sim A_2$, if every step performed by A_1 is simulated by A_2 , and vice versa, in the same order and with the same probability. Note that a single step is associated with both externally visible and internal step, hence the conditions for strong bisimulation for PLTS can also be applied to PCA as no distinction is needed due to the introduction of action types. A strong bisimulation for PCA models is then defined using an equivalence relationship $\mathcal{R} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}: \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \hat{\mathcal{E}}$:

$$\hat{\mu}_G(s, a, C) = \hat{\mu}_G(t, a, C). \quad (\text{A.36})$$

Equivalently to the case of PLTS models, two PCA models are strongly bisimilar iff there is a strong bisimulation $\mathcal{R}$ on $\hat{A}$ such that $(q_1, q_2) \in \mathcal{R}$. In the following paragraphs we will illustrate the application of bisimulation rules to two example PCA models.

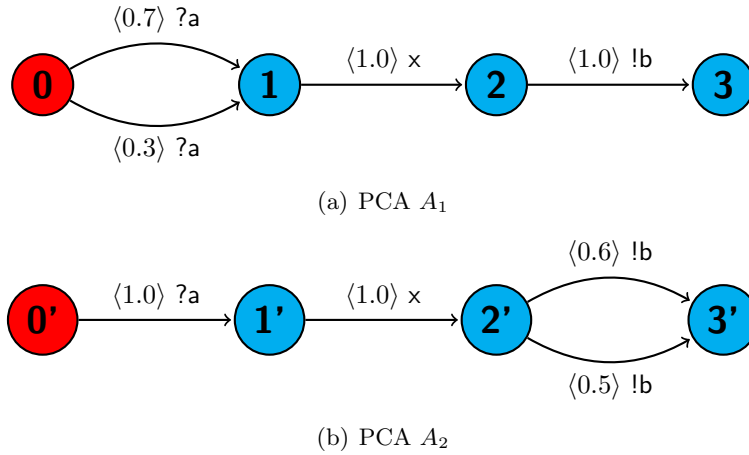


Figure A.5.: Examples of PCA Models for Strong Bisimulation

Consider the example PCA models A_1 and A_2 in Figure A.5 and their disjoint union $\hat{A}$ in Figure A.8. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and A_2 : $\{(0, 0'), (1, 1'), (2, 2'), (3, 3')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{1, 1'\}, \{2, 2'\}, \{3, 3'\}\}$. Using the strong bisimulation definition in equation A.36, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a strong bisimulation.

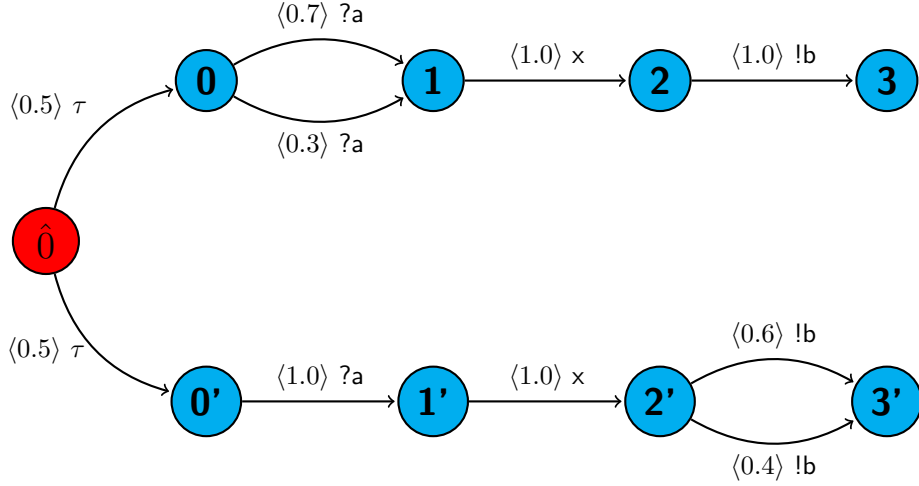


Figure A.6.: Disjoint Union PCA $\hat{A}$ for Strong Bisimulation Between PCA Models A_1 and A_2

We start with the case where $(s, t) = (0, 0')$ and $C = \{0, 0'\}$ for the action label a :

$$\hat{\mu}_G(0, a, \{0, 0'\}) = \hat{\mu}_G(0', a, \{0, 0'\}). \quad (\text{A.37})$$

By applying the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$\hat{\mu}(0, a, 0) + \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) + \hat{\mu}(0', a, 0'). \quad (\text{A.38})$$

Given that the transitions $(0, a, 0)$, $(0, a, 0')$, $(0', a, 0)$ and $(0', a, 0')$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}(0, a, 0) = \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) = \hat{\mu}(0', a, 0') = 0, \quad (\text{A.39})$$

and we conclude that the condition for strong bisimulation is verified as follows for the current case

$$\hat{\mu}(0, a, 0) + \hat{\mu}(0, a, 0') = \hat{\mu}(0', a, 0) + \hat{\mu}(0', a, 0') \iff 0 + 0 = 0 + 0 \quad \square \quad (\text{A.40})$$

We omit the cases where $(s, t) = (1, 1')$ and $C = \{1, 1'\}$ as well as $(s, t) = (1, 1')$ and $C = \{0, 0'\}$ for the action label a as these are equivalent to the previous case since there are no transitions between the bisimilar states and the states in the equivalence classes.

Consider now the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the action label a :

$$\hat{\mu}_G(0, a, \{1, 1'\}) = \hat{\mu}_G(0', a, \{1, 1'\}). \quad (\text{A.41})$$

By applying the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$\hat{\mu}_G(0, a, 1) + \hat{\mu}_G(0, a, 1') = \hat{\mu}_G(0', a, 1) + \hat{\mu}_G(0', a, 1'). \quad (\text{A.42})$$

Given that the transitions $\hat{\mu}_G(0, a, 1')$ and $\hat{\mu}_G(0', a, 1)$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}_G(0, a, 1) + 0 = 0 + \hat{\mu}_G(0', a, 1'). \quad (\text{A.43})$$

By replacing $\hat{\mu}_G(0, a, 1)$ by $\mu_{1G}(0, a, 1)$ and $\hat{\mu}_G(0', a, 1')$ by $\mu_{2G}(0', a, 1')$ we confirm that the condition for strong bisimulation is also verified for this case:

$$0.7 + 0.3 = 1 \quad \square \quad (\text{A.44})$$

We omit the following cases as their verification is analogous to the previous case:

- $(s, t) = (1, 1')$ and $C = \{2, 2'\}$ for the action label x ;
- $(s, t) = (2, 2')$ and $C = \{3, 3'\}$ for the action label b .

The equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for strong bisimulation defined by equation A.36, includes the initial states 0 and 0' of A_1 and A_2 , respectively. As a consequence, the PCA models A_1 and A_2 are strongly bisimilar.

A.3.2. Weak Bisimulation

While weak bisimulation for PLTS models is defined based on the probability of paths from bisimilar states s and s' which contain action a and may be preceded and/or followed by transitions labelled with internal action τ , in PCA internal steps are labelled with internal actions, as opposed to τ . As a result, weak bisimulation for PCA models is defined based on the probability of paths from bisimilar states s and s' which contain action a *visible* a and may be preceded

and/or followed by transitions labelled with internal actions. We define the set of visible actions $\mathcal{E}^{visible}$ as the union of input, output and failure actions: $\mathcal{E}^{visible} = \mathcal{E}^{in} \cup \mathcal{E}^{out} \cup \mathcal{E}^{fail}$. Based on the extension of the function μ_G for a PCA A to sequences of actions in $\mathcal{E}^*$, we define the notion of weak bisimulation for PCA models through an equivalence relationship $\mathcal{R} \subseteq \hat{\mathcal{S}} \times \hat{\mathcal{S}}$ such that $\forall (s, t) \in \mathcal{R}, \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \mathcal{E}^{visible}$:

$$\hat{\mu}_G(s, \tau^* a \tau^*, C) = \hat{\mu}_G(t, \tau^* a \tau^*, C), \quad (\text{A.45})$$

$$\hat{\mu}_G(s, \tau^*, C) = \hat{\mu}_G(t, \tau^*, C), \text{ where } \tau \in \mathcal{E}^{int}. \quad (\text{A.46})$$

This condition determines that two states s and t are weakly bisimilar if the sum of the probability across all the paths over a sequence of zero or more internal actions followed by a visible action a and a sequence of zero or more internal actions between them, and states in the equivalence class C is the same. Note that the paths between state s and states in C may not have the same length as the corresponding paths between state t and C . Formally, two PLTS models A_1 and A_2 are weakly bisimilar iff there is a weak bisimulation $\mathcal{R}$ on $\hat{A}$ such that $(q_1, q_2) \in \mathcal{R}$.

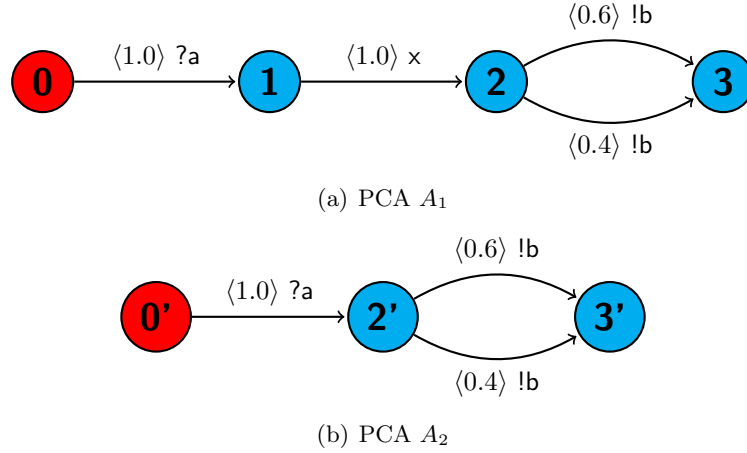


Figure A.7.: Examples of PCA Models for Weak Bisimulation

Consider the example PCA models A_1 and A_2 in Figure A.7 and their disjoint union $\hat{A}$ in Figure A.8. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and A_2 : $\{(0, 0'), (1, 1'), (3, 3')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{1, 1'\}, \{3, 3'\}\}$. Using the weak bisimulation definition in equation A.45, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a weak bisimulation.

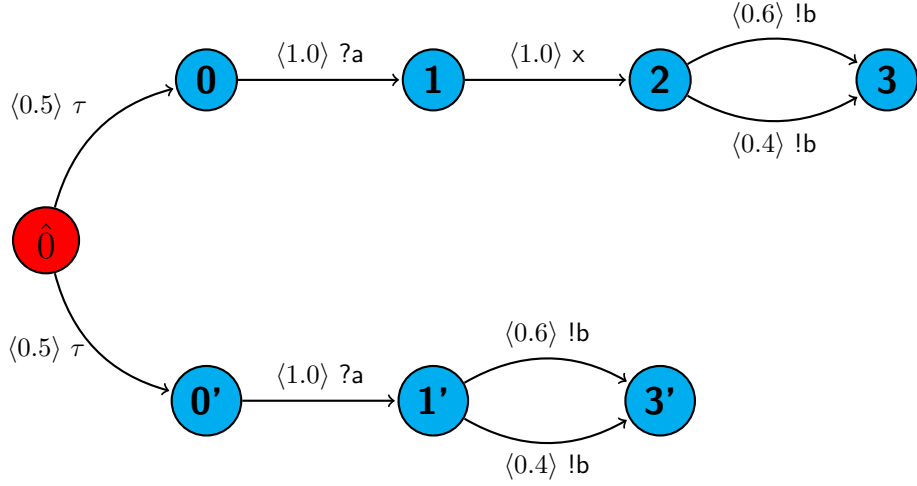


Figure A.8.: Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2

We start by considering the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the input action a :

$$\hat{\mu}_G(0, \tau^* ?a \tau^*, \{1, 1'\}) = \hat{\mu}_G(0', \tau^* ?a \tau^*, \{1, 1'\}). \quad (\text{A.47})$$

Given that there is a single path between states 0 and 1 as well as between states $0'$ and $1'$ we derive the following from the expansion rule A.19:

$$\hat{\mu}_G(0, \epsilon, 0) \cdot \hat{\mu}_G(0, ?a, \{1, 1'\}) = \hat{\mu}_G(0', \epsilon, 0') \cdot \hat{\mu}_G(0', \tau^* ?a \tau^*, \{1, 1'\}). \quad (\text{A.48})$$

By applying rule A.17 to $\hat{\mu}_G(0, \epsilon, 0)$ and $\hat{\mu}_G(0', \epsilon, 0')$ as well as the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$1 \cdot \left(\hat{\mu}_G(0, ?a, 1) + \hat{\mu}_G(0, ?a, 1') \right) = 1 \cdot \left(\hat{\mu}_G(0', a, 1) + \hat{\mu}_G(0', a, 1') \right). \quad (\text{A.49})$$

Given that the transitions $\hat{\mu}_G(0, ?a, 1')$ and $\hat{\mu}_G(0', ?a, 1)$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}_G(0, ?a, 1) + 0 = 0 + \hat{\mu}_G(0', ?a, 1'). \quad (\text{A.50})$$

By replacing $\hat{\mu}_G(0, ?a, 1)$ by $\mu_{1G}(0, ?a, 1)$ and $\hat{\mu}_G(0', ?a, 1')$ by $\mu_{2G}(0', ?a, 1')$ we confirm that the condition for weak bisimulation is verified for this case:

$$1 = 1 \quad \square \quad (\text{A.51})$$

Consider now the case where $(s, t) = (1, 1')$ and $C = \{3, 3'\}$ for the output action $!b$:

$$\hat{\mu}_G(1, \tau^* !b \tau^*, \{3, 3'\}) = \hat{\mu}_G(1', \tau^* !b \tau^*, \{3, 3'\}). \quad (\text{A.52})$$

While there is only one transition in the path between states $1'$ and $3'$ in $\hat{A}$, the paths between states 1 and 3 include transitions through intermediate state 2 . Therefore, the general condition A.21 for weak bisimulation is mapped to:

$$\hat{\mu}_G(1, \tau !b, \{3, 3'\}) = \hat{\mu}_G(1', !b, \{3, 3'\}). \quad (\text{A.53})$$

Using the expansion rule A.19 on $\hat{\mu}_G(1, \tau !b, \{3, 3'\})$ we obtain:

$$\sum_{s \in \hat{\mathcal{S}}} \hat{\mu}_G(1, \tau, s) \cdot \hat{\mu}_G(s, !b, \{3, 3'\}) = \hat{\mu}_G(1', !b, \{3, 3'\}). \quad (\text{A.54})$$

Given that there is only a single transition from state 1 to state 2 , then for all states $s \in \hat{\mathcal{S}} \setminus \{2\}$, $\hat{\mu}_G(1, \tau, s) \cdot \hat{\mu}_G(s, !b, \{3, 3'\}) = 0$ as $\hat{\mu}_G(1, \tau, s) = 0$. As only the transition between states 1 and 2 has a probability greater than 0 , we obtain the following by also applying the rule A.7 when μ_G is applied to a set of states:

$$\hat{\mu}_G(1, \tau, 2) \cdot (\hat{\mu}_G(2, !b, 3) + \hat{\mu}_G(2, !b, 3')) = \hat{\mu}_G(1', !b, 3) + \hat{\mu}_G(1', !b, 3'). \quad (\text{A.55})$$

By replacing $\hat{\mu}_G$ by the corresponding cumulative probability functions from A_1 and A_2 we derive

$$\mu_{1G}(1, \tau, 2) \cdot (\mu_{1G}(2, !b, 3) + 0) = 0 + \mu_{2G}(1', !b, 3'). \quad (\text{A.56})$$

and confirm that the condition for weak bisimulation is also verified for this case:

$$1 \cdot (0.6 + 0.4) = (0.6 + 0.4) \quad \square \quad (\text{A.57})$$

The equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for weak bisimulation defined by equation A.45 and includes the initial states 0 and $0'$ of A_1 and A_2 , respectively. As a consequence, the PCA models A_1 and A_2 are weakly bisimilar.

A.4. Hiding Operator

In the previous Section we defined notions of strong and weak bisimulation for PCA models, given any two models A_1 and A_2 . In this Section we show that the reduced model A_2 produced by the hiding operator when applied to an original model A_1 is weakly bisimilar to A_1 . We start by reviewing the steps performed by the reduction algorithm when producing the model A_2 .

The hiding operator is implemented using the reduction algorithm presented in Chapter 4; the input model A_1 is traversed and the incoming/outgoing transitions of each state s are analysed according to the two main cases considered by the algorithm to determine which ones can be deleted and whether probabilities can be propagated to subsequent or prior transitions. We formalise each case as a transformation T_i that is applied to any state of an input PCA A_1 and produces a reduced PCA A_2 along with an equivalence relationship $\mathcal{R}_{T_i}$. We then show that the models A_1 and A_2 are weakly bisimilar by showing that the equivalence relationship $\mathcal{R}_{T_i}$ is a weak bisimulation that contains the initial states of A_1 and A_2 . Moreover, the reduction algorithm can be seen as a sequence of transformations $\mathcal{R}_{T_i}$ applied to the original model A_1 . Therefore, we show that the model A_2 produced of the reduction algorithm is weakly bisimilar to the original model A_1 by showing that the relationships $\mathcal{R}_{T_i}$ are transitive, *i.e.* any sequence of transformations produces a weakly bisimilar model.

A.4.1. Preliminaries

The different cases considered by the reduction algorithm take as input a PCA $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1, \mu_1 \rangle$, a sub-set of internal actions $\mathcal{E}_1^R \subseteq \mathcal{E}_1^{int}$ and a state $s_{src} \in \mathcal{S}_1$. Each case is modelled as a transformation T_i which takes as input the tuple $\langle A_1, \mathcal{E}_1^R, s_{src} \rangle$ and produces a (reduced) PCA $A_2 = \langle \mathcal{S}_2, q_2, \mathcal{E}_2, \Delta_2, \mu_2 \rangle$ and a bisimulation that establishes an equivalence relationship $\mathcal{R}_{T_i}$ between states in $\mathcal{S}_1$ and $\mathcal{S}_2$. In the next Sections we discuss how each transformation constructs the PCA A_2 from an input PCA A_1 and the corresponding equivalence relationship $\mathcal{R}_{T_i}$. We then show that $\mathcal{R}_{T_i}$ is a weak bisimulation.

A.4.2. Transformation T_1

Transformation T_1 represents the first case of the reduction algorithm which analyses a single transition $(s_{src}, a_r, s_{dst}) \in \Delta_1$ labelled with action $a_r \in \mathcal{E}_1^R$ and determines if the transition is removed when constructing A_2 from A_1 . The transition (s_{src}, a_r, s_{dst}) is removed if the following conditions hold:

- the transition (s_{src}, a_r, s_{dst}) is the only incoming transition to state s_{dst} and the outgoing transitions of state s_{dst} are not labelled with input/output actions: $\#(\rho(s')) = 1 \wedge \Delta_1^e(s_{dst}) \cap \mathcal{E}_1^{in} = \emptyset \wedge \Delta_1^e(s_{dst}) \cap \mathcal{E}_1^{out} = \emptyset$;
- the transition (s_{src}, a_r, s_{dst}) is both the only incoming transition to state s_{dst} and the only outgoing transition of state s_{src} which implies $\mu_1(s_{src}, a_r, s_{dst}) = 1$;

When the transition (s_{src}, a_r, s_{dst}) is deleted, the PCA A_2 and the equivalence relationship $\mathcal{R}_{T_1}$ are defined as follows. For all transitions $(s, a, s') \in \Delta_1$, except the transition $\{(s_{src}, a_r, s_{dst})\}$, if the source state s is not the destination state s_{dst} of the deleted transition, the following steps as performed:

- state s is mapped to a state t and (s, t) is added to $\mathcal{R}_{T_1}$;
- state s' is mapped to a state t' and (s', t') is added to $\mathcal{R}_{T_1}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \mu_1(s, a, s'). \quad (\text{A.58})$$

When the transition source state s is the destination state s_{dst} , the probability of the deleted transition (s_{src}, a_r, s_{dst}) is propagated to the outgoing transitions of s_{dst} $\left((s_{dst}, a, s')\right)$ as follows:

- state s_{src} is mapped to a state t and (s_{src}, t) is added to $\mathcal{R}_{T_1}$;
- state s' is mapped to a state t' and (s', t') is added to $\mathcal{R}_{T_1}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_1(s_{dst}, a, s'). \quad (\text{A.59})$$

The state s_{dst} is not mapped to any other state in $\mathcal{S}_2$ as this state is merged with state s_{src} when constructing the PCA model A_2 . As a result, for each state $s \in \mathcal{S}_1 \setminus \{s_{dst}\}$ there exists one and only one state $t \in \mathcal{S}_2$ such that $(s, t) \in \mathcal{R}_{T_1}$:

$$[s]_{\mathcal{R}_{T_1}} = \{t\} \wedge [t]_{\mathcal{R}_{T_1}} = \{s\}. \quad (\text{A.60})$$

Consequently, all equivalence classes $C \in \hat{\mathcal{S}} / \mathcal{R}_{T_1}$ have two states $\{s, t\}$, where $s \in \mathcal{S}_1$ and $t \in \mathcal{S}_2$. Finally, the set of action labels $\mathcal{E}_2$ is equal to $\mathcal{E}_1 \setminus \{a_r\}$ if (s_{src}, a_r, s_{dst}) is the only transition in A_1 labelled with a_r ; and equal to $\mathcal{E}_1$ otherwise.

In order for $\mathcal{R}_{T_1}$ to be a weak bisimulation on the disjoint union $\hat{A}$ of A_1 and A_2 , the following conditions have to hold $(q_1, q_2) \in \mathcal{R}_{T_1} \wedge \forall (s, t) \in \mathcal{R}_{T_1}, \forall C \in \hat{\mathcal{S}}_C, \forall a \in \hat{\mathcal{E}}^{visible}$:

$$\hat{\mu}_G(s, (a^{int})^* a (a^{int})^*, C) = \hat{\mu}_G(t, (a^{int})^* a (a^{int})^*, C), \quad (\text{A.61})$$

$$\hat{\mu}_G(s, (a^{int})^*, C) = \hat{\mu}_G(t, (a^{int})^*, C), \text{ where } a^{int} \in \hat{\mathcal{E}}^{int}. \quad (\text{A.62})$$

If the transition (s_{src}, a_r, s_{dst}) is not deleted, then A_2 is strongly bisimilar to A_1 and is therefore also weakly bisimilar. When the transition (s_{src}, a_r, s_{dst}) is deleted we consider the following two cases to show that $\mathcal{R}_{T_1}$ is a weak bisimulation. Although for simplicity we assume that for each $(s, t) \in \mathcal{R}_{T_1}$, state s belongs to A_1 and state t belongs to A_2 , the demonstration for the reciprocal case is equivalent. Consequently, we will demonstrate that $(q_1, q_2) \in \mathcal{R}_{T_1} \wedge \forall (s, t) \in \mathcal{R}_{T_1}, \forall C \in \hat{\mathcal{S}}_C, \forall a \in \hat{\mathcal{E}}^{visible}$:

$$\mu_{1G}(s, (a^{int})^* a (a^{int})^*, C) = \mu_{2G}(t, (a^{int})^* a (a^{int})^*, C), \quad (\text{A.63})$$

$$\mu_{1G}(s, (a^{int})^* a (a^{int})^*, C) = \mu_{2G}(t, (a^{int})^*, C). \quad (\text{A.64})$$

Given that the equivalences $C \in \hat{\mathcal{S}}_C$ are of the form $\{s', t'\}$, we further assume state s' belongs to A_1 and state t' belongs to A_2 . The following condition is then derived from equations A.63 and A.64:

$$\mu_{1G}(s, (a^{int})^* a (a^{int})^*, \{s', t'\}) = \mu_{2G}(t, (a^{int})^* a (a^{int})^*, \{s', t'\}), \quad (\text{A.65})$$

$$\mu_{1G}(s, (a^{int})^*, \{s', t'\}) = \mu_{2G}(t, (a^{int})^*, \{s', t'\}) \quad (\text{A.66})$$

Using rule A.7, we expand the first equation as follows:

$$\mu_{1G}(s, (a^{int})^* a (a^{int})^*, s') + \mu_{1G}(s, (a^{int})^* a (a^{int})^*, s', t') \quad (\text{A.67})$$

$$= \mu_{2G}(t, (a^{int})^* a (a^{int})^*, s') + \mu_{2G}(t, (a^{int})^* a (a^{int})^*, t'). \quad (\text{A.68})$$

As there are no transitions from state s to state t' as well as from state t to state s' in A_1 and A_2 , respectively, we obtain:

$$\mu_{1G}(s, (a^{int})^* a (a^{int})^*, s') = \mu_{2G}(t, (a^{int})^* a (a^{int})^*, t'), \quad (\text{A.69})$$

$$\mu_{1G}(s, (a^{int})^*, s') = \mu_{2G}(t, (a^{int})^*, t'). \quad (\text{A.70})$$

While the above conditions cover all possible sequences of internal actions as a prefix and suffix of a visible action a , as well as sequences of internal actions, we consider the application of condition A.69 to a particular sequence $\lambda = a_1^{int} \dots a_{l_1}^{int} a b_{l_2}^{int} \dots b_{l_2}^{int}$, where $\forall_{i=1}^{l_1} a_i \in \hat{\mathcal{E}}^{int}$ and $\forall_{i=1}^{l_2} b_i \in \hat{\mathcal{E}}^{int}$. In order to verify the condition A.69, it needs to be validated for the following cases:

- the transition (s_{src}, a_r, s_{dst}) is not present in any path based on sequence λ between states s and s' ;
- the transition (s_{src}, a_r, s_{dst}) is the first transition in the path based on sequence λ between states s and s' , hence $s = s_{src}$ and $a_r = a_1^{int}$;
- the transition (s_{src}, a_r, s_{dst}) is present in the path based on sequence λ between states s and s' , but it is not the last transition of the first and last part, thus $a_r \neq a_{l_1}$ and $a_r \neq b_{l_2}$;
- the transition (s_{src}, a_r, s_{dst}) is the last transition in the first part of the path based on sequence λ between states s and s' , hence $s_{l_1} = s_{src}$ and $a_r = a_{l_1}^{int}$.

Case 1:

We first consider the case where the transition (s_{src}, a_r, s_{dst}) is not included in any path between states s and s' . As a result, the sequence λ is applied to both μ_{1G} and μ_{2G} :

$$\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda, t'). \quad (\text{A.71})$$

Following rule A.19, we expand $\mu_{1G}(s, \lambda, s')$ as follows:

$$\mu_{1G}(s, \lambda, s') = \sum_{s_1 \in \mathcal{S}_1} \mu_{1G}(s, a_1^{int}, s_1) \cdot \mu_{1G}(s_1, \lambda', s'), \text{ where } \lambda = a_1^{int} \lambda'. \quad (\text{A.72})$$

We derive the general expansion as follows. Consider l_1 as the length of prefix sequence of internal actions $a_1^{int} \dots a_{l_1}^{int}$ that precedes the visible action label a and l_2 as the length of the

sequence of internal actions $b_1^{int} \dots b_{l_2}^{int}$ that succeeds the visible action label a , the full expansion of $\mu_{1G}(s, \lambda, s')$ is given as follows:

$$\begin{aligned} \mu_{1G}(s, \lambda, s') &= \sum_{s_1 \in \mathcal{S}_1} \left(\mu_{1G}(s, a_1^{int}, s_1) \cdot \prod_{i=2}^{l_1} \left(\sum_{s_i \in \mathcal{S}_1} \mu_{1G}(s_{i-1}, a_i^{int}, s_i) \right) \right. \\ &\quad \left. \left(\sum_{s'_1 \in \mathcal{S}_1} \mu_{1G}(s_{l_1}, a, s'_1) \cdot \prod_{i=2}^{l_2} \left(\sum_{s'_i \in \mathcal{S}_1} \mu_{1G}(s'_{i-1}, b_{i-1}^{int}, s'_i) \right) \cdot \mu_{1G}(s'_{l_2}, b_{l_2}^{int}, s') \right) \right) \end{aligned} \quad (\text{A.73})$$

As the transition (s_{src}, a_r, s_{dst}) is not included in any path between states s and s' , $\mu_{2G}(t, \lambda, t')$ is similarly expanded as follows:

$$\begin{aligned} \mu_{2G}(t, \lambda, t') &= \sum_{t_1 \in \mathcal{S}_2} \left(\mu_{2G}(t, a_1^{int}, t_1) \cdot \prod_{i=2}^{l_1} \left(\sum_{t_i \in \mathcal{S}_2} \mu_{2G}(t_{i-1}, a_i^{int}, t_i) \right) \right. \\ &\quad \left. \left(\sum_{t'_1 \in \mathcal{S}_2} \mu_{2G}(t_{l_1}, a, t'_1) \cdot \prod_{i=2}^{l_2} \left(\sum_{t'_i \in \mathcal{S}_2} \mu_{2G}(t_{i-1}, b_{i-1}^{int}, t'_i) \right) \cdot \mu_{2G}(t'_{l_2}, b_{l_2}^{int}, t') \right) \right) \end{aligned} \quad (\text{A.74})$$

Given rule A.58, for each state $s_i \in \mathcal{S}_1$ in the paths with the sequence λ , there is a corresponding state $t_i \in \mathcal{S}_2$ such that $(s_i, t_i) \in \mathcal{R}_{T_i}$. Therefore, $\forall_{i=1}^{l_1-1} \mu_{1G}(s_i, a_i^{int}, s_{i+1}) = \mu_{2G}(t_i, a_i^{int}, t_{i+1})$. Similarly, for each state $s'_i \in \mathcal{S}_1$ in the paths with the sequence λ , there is a corresponding state $t'_i \in \mathcal{S}_2$ such that $(s'_i, t'_i) \in \mathcal{R}_{T_i}$. Consequently, $\forall_{i=2}^{l_2} \mu_{1G}(s'_{i-1}, b_{i-1}^{int}, s'_i) = \mu_{2G}(t'_{i-1}, b_{i-1}^{int}, t'_i)$. The sample principle applies to the transitions $(s'_{l_2}, b_{l_2}^{int}, s')$ and $(t'_{l_2}, b_{l_2}^{int}, t')$ as both (s'_{l_2}, t'_{l_2}) and (s', t') belong to the equivalence relationship $\mathcal{R}_{T_1}$, thus $\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda, t')$ in this case.

Case 2:

We now consider the case where the deleted transition is the first in the sequence $\lambda = a_1^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}$, hence $s = s_{src}$ and $a_1^{int} = a_r$. Therefore, we verify the following conditions

$$\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda', t'), \text{ where } \lambda = a_r \lambda', \quad (\text{A.75})$$

as transition (s_{src}, a_r, s_{dst}) has not been mapped to A_2 . We start by following rule A.19 to expand $\mu_{1G}(s, \lambda, s')$. Note that in order for transition (s_{src}, a_r, s_{dst}) to be deleted it needs to

be the only incoming transition to state s_{dst} . As a result, state s_{dst} is only reachable through transition (s_{src}, a_r, s_{dst}) :

$$\mu_{1G}(s, \lambda, s') = \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_{dst}, \lambda', s'). \quad (\text{A.76})$$

We further expand the above expression to

$$\mu_{1G}(s, \lambda, s') = \mu_1(s_{src}, a_r, s_{dst}) \cdot \left(\sum_{s_2 \in \mathcal{S}_1} \mu_{1G}(s_{dst}, a_2^{int}, s_2) \cdot \mu_{1G}(s_2, \lambda'', s') \right), \text{ where } \lambda = a_r a_2^{int} \lambda'', \quad (\text{A.77})$$

which is equivalent to

$$\mu_{1G}(s, \lambda, s') = \left(\sum_{s_2 \in \mathcal{S}_1} \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_{dst}, a_2^{int}, s_2) \cdot \mu_{1G}(s_2, \lambda'', s') \right). \quad (\text{A.78})$$

On the other hand, $\mu_{2G}(t, \lambda', t')$ is expanded as follows:

$$\mu_{2G}(t, \lambda', t') = \sum_{t_2 \in \mathcal{S}_2} \mu_{2G}(t, a_2^{int}, t_2) \cdot \mu_{2G}(t_2, \lambda'', s'). \quad (\text{A.79})$$

Given rule A.59 for constructing the PCA A_2 from A_1 by applying transformation T_1 , $\forall s_2 \in \mathcal{S}_1$ and $\forall t_2 \in \mathcal{S}_2$ such that $(s_2, t_2) \in \mathcal{R}_{T_1}$, then

$$\mu_{2G}(t, a_2^{int}, t_2) = \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_{dst}, a_2^{int}, s_2). \quad (\text{A.80})$$

We have shown in the previous case that $\forall s_2 \in \mathcal{S}_1$ and $\forall t_2 \in \mathcal{S}_2$ such that $(s_2, t_2) \in \mathcal{R}_{T_1}$, then

$$\mu_{1G}(s_2, \lambda'', s') = \mu_{2G}(t_2, \lambda'', s'). \quad (\text{A.81})$$

Therefore, $\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda', t')$ when the deleted transition is the first in the sequence $\lambda = a_1^{int} \dots a_{l_1}^{int} a a_{l_1'}^{int} \dots a_{l_2}^{int}$.

Case 3:

We now consider the general case where the deleted transition is between states s and s' but is not the last transition in the sequence $\lambda = a_1^{int} \dots a_{i-1}^{int} a_i^{int} a_{i+1}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}$. Therefore, we verify the following conditions

$$\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda', t'), \text{ where } \lambda' = a_1^{int} \dots a_{i-1}^{int} a_{i+1}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}. \quad (\text{A.82})$$

We first apply the expansion rule A.19 to $\mu_{1G}(s, \lambda, s')$ and obtain

$$\begin{aligned} \mu_{1G}(s, \lambda, s') &= \mu_{1G}(s, a_1^{int} \dots a_{i-1}^{int}, s_{i-1}) \cdot \mu_{1G}(s_{i-1}, a_i^{int}, s_i) \cdot \\ &\quad \mu_{1G}(s_i, a_{i+1}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, s'), \end{aligned} \quad (\text{A.83})$$

where the deleted transition $(s_{src}, a_r, s_{dst}) = (s_{i-1}, a_i^{int}, s_i)$. Note that as we are only considering paths that include the deleted transition, we do not need to consider paths through all possible states $s_{i-1} \in \mathcal{S}_1$ when applying the expansion rule to $\mu_{1G}(s, \lambda, s')$ as state s_i can only be reached through the deleted transition, according the rules of transformation T_1 . We further expand $\mu_{1G}(s, \lambda, s')$ into

$$\begin{aligned} \mu_{1G}(s, \lambda, s') &= \mu_{1G}(s, a_1^{int} \dots a_{i-1}^{int}, s_{i-1}) \cdot \mu_1(s_{src}, a_r, s_{dst}) \cdot \\ &\quad \left(\sum_{s_{i+1} \in \mathcal{S}_1} \mu_{1G}(s_i, a_{i+1}, s_{i+1}) \cdot \mu_{1G}(s_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, s') \right), \end{aligned} \quad (\text{A.84})$$

which is equivalent to

$$\begin{aligned} \mu_{1G}(s, a_1^{int} \dots a_{i-1}^{int}, s_{i-1}) &\cdot \left(\sum_{s_{i+1} \in \mathcal{S}_1} \mu_1(s_{src}, a_r, s_{dst}) \cdot \right. \\ &\quad \left. \mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1}) \cdot \mu_{1G}(s_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, s') \right). \end{aligned} \quad (\text{A.85})$$

By applying the same steps to $\mu_{2G}(t, \lambda', t')$ we obtain:

$$\mu_{2G}(t, a_1^{int} \dots a_{i-1}^{int}, t_{i-1}) \cdot \left(\sum_{t_{i+1} \in \mathcal{S}_1} \mu_{2G}(t_{i-1}, a_{i+1}^{int}, t_{i+1}) \cdot \mu_{2G}(t_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, t') \right). \quad (\text{A.86})$$

We have shown in the previous cases that $\mu_{1G}(s, a_1^{int} \dots a_{i-1}^{int}, s_{i-1}) = \mu_{2G}(t, a_1^{int} \dots a_{i-1}^{int}, t_{i-1})$ and $\mu_{1G}(s_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, s') = \mu_{2G}(t_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, t')$. Based on the rules of transformation T_1 , we can derive that $(s_{i-1}, t_{i-1}) \in \mathcal{R}_{T_1} \wedge (s_{i+1}, t_{i+1}) \in \mathcal{R}_{T_1}$ and

$$\mu_{2G}(t_{i-1}, a_{i+1}^{int}, t_{i+1}) = \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1}) . \quad (\text{A.87})$$

Case 4:

The case where the deleted transition is in the second part of the sequence λ after visible action a is equivalent to the case we just discussed. Similarly, the case where the deleted transition (s_{src}, a_r, s_{dst}) is the last transition in the first path that precedes the visible action a in the sequence $\lambda = a_1^{int} \dots a_{l_1-1}^{int} a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}$ is also equivalent to the previous case as the visible action needs to be an output action for the transition s_{src}, a_r, s_{dst} and its probability to be propagated.

Furthermore, the verification of condition A.70 ($\mu_{1G}(s, (a^{int})^*, s') = \mu_{2G}(t, (a^{int})^*, t')$) is similar to the verification steps applied in the previous paragraphs to the prefix sequence $a_{l_1-1}^{int} a_{l_1}^{int}$. We have therefore shown that transformation T_1 produces a weakly bisimilar model as defined by the equivalence relationship $\mathcal{R}_{T_1}$.

Example

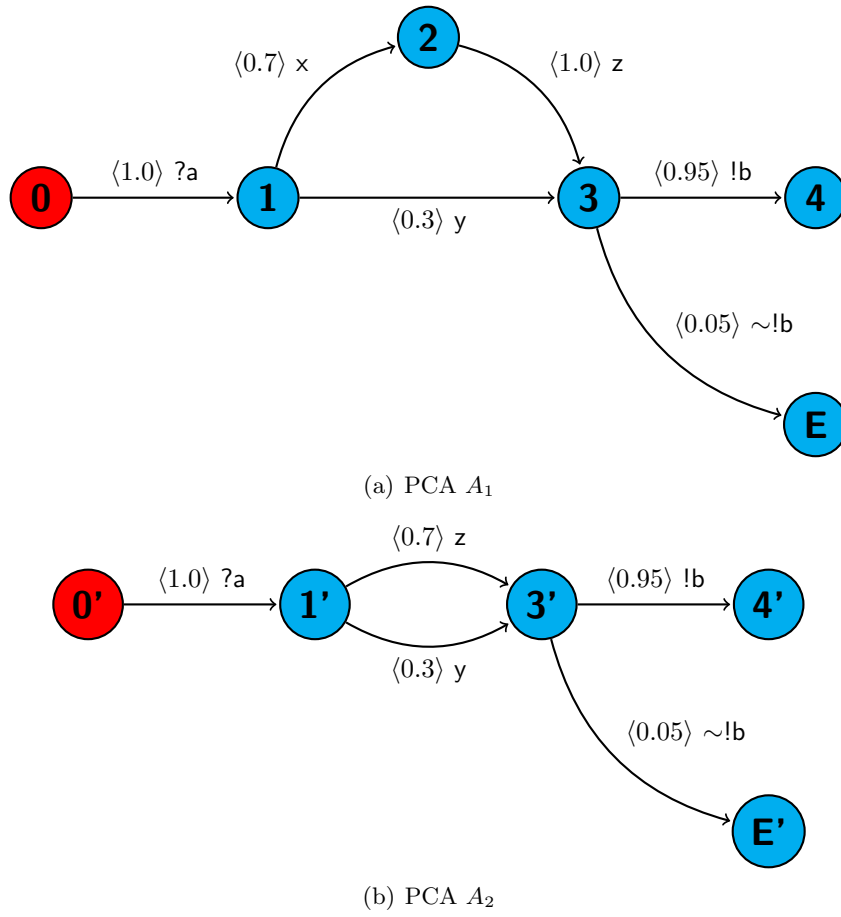
We illustrate in the next paragraphs the application of the above steps to the example PCA models A_1 and A_2 in Figure A.9 and their disjoint union $\hat{A}$ in Figure A.10. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and $A_2 : \{(0, 0'), (1, 1'), (3, 3'), (4, 4'), (E, E')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{1, 1'\}, \{3, 3'\}, \{4, 4'\}, \{E, E'\}\}$. Using the weak bisimulation definition in equation A.45, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a weak bisimulation.

We start by considering the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the input action a :

$$\hat{\mu}_G(0, \tau^* ?a \tau^*, \{1, 1'\}) = \hat{\mu}_G(0', \tau^* ?a \tau^*, \{1, 1'\}). \quad (\text{A.88})$$

By applying the expansion rules A.7, A.17 and A.19 we obtain

$$\hat{\mu}_G(0, ?a, 1) = \hat{\mu}_G(0', ?a, 1'). \quad (\text{A.89})$$

Figure A.9.: Examples of PCA Models for Weak Bisimulation - Transformation T_1

By replacing $\hat{\mu}_G(0, ?a, 1)$ by $\mu_{1G}(0, ?a, 1)$ and $\hat{\mu}_G(0', ?a, 1')$ by $\mu_{2G}(0', ?a, 1')$ we confirm that the condition for weak bisimulation is verified for this case:

$$1 = 1 \quad \square \tag{A.90}$$

The case where $(s, t) = (0, 0')$ and $C = \{3, 3'\}$ for the input action $?a$ can be verified as follows by applying the expansion rules A.7, A.17 and A.19 to the condition A.45:

$$\mu_{1G}(0, ?a, 1) \cdot \mu_{1G}(1, \tau^*, 3) = \mu_{2G}(0', ?a, 1') \cdot \mu_{2G}(1', \tau^*, 3'). \tag{A.91}$$

We further expand the above expressions and obtain

$$\mu_{1G}(0, ?a, 1) \cdot (\mu_{1G}(1, \tau, 2) \cdot \mu_{1G}(2, \tau, 3) + \mu_{1G}(1, \tau, 3)) = \mu_{2G}(0', ?a, 1') \cdot \mu_{2G}(1', \tau, 3'). \tag{A.92}$$

By replacing μ_{1G} and μ_{2G} by the probabilities of the corresponding transitions we verify the condition A.91:

$$1 \cdot (0.7 \cdot 1 + 0.3) = 1 \cdot (0.7 + 0.3) \quad \square. \tag{A.93}$$

We omit the illustration for the case where $(s, t) = (1, 1')$ and $C = \{4, 4'\}$ for the output action $!b$ as it can be verified using the same steps as for the case where $(s, t) = (0, 0')$ and $C = \{1, 1'\}$ for the input action a . We analyse instead the case where $(s, t) = (1, 1')$ and $C = \{4, 4'\}$ for the output action $!b$:

$$\hat{\mu}_G(1, \tau^* !b \tau^*, \{4, 4'\}) = \hat{\mu}_G(1', \tau^* !b \tau^*, \{4, 4'\}). \tag{A.94}$$

Given that the paths from state 1 to 4 and from state $1'$ to $4'$ do not include an internal transition after transitions labelled with output action $!b$, we simplify the previous conditions to

$$\hat{\mu}_G(1, \tau^* !b, \{4, 4'\}) = \hat{\mu}_G(1', \tau^* !b, \{4, 4'\}). \tag{A.95}$$

Using the expansion rule A.19 on $\hat{\mu}_G(1, \tau !b, \{3, 3'\})$ we obtain:

$$\hat{\mu}_G(1, \tau^*, 3) \cdot \hat{\mu}_G(3, !b, \{4, 4'\}) = \hat{\mu}_G(1', \tau^*, 3') \cdot \hat{\mu}_G(3', !b, \{4, 4'\}). \tag{A.96}$$

By replacing $\hat{\mu}_G$ by the corresponding cumulative probability functions from A_1 and A_2 and further expanding the sequences prefix τ^* we derive

$$\left(\mu_{1G}(1, \tau, 2) \cdot \mu_{1G}(2, \tau, 3) + \mu_{1G}(1, \tau, 3) \right) \cdot \mu_{1G}(3, !b, \{4, 4'\}) = \mu_{2G}(1', \tau, 3') \cdot \mu_{2G}(3', !b, \{4, 4'\}). \quad (\text{A.97})$$

and confirm that the condition A.103 for weak bisimulation is also verified for this case:

$$(0.7 \cdot 1 + 0.3) \cdot 0.95 = (0.7 + 0.3) \cdot 0.95 \quad \square. \quad (\text{A.98})$$

The verification of paths involving output action $\sim!b$ are verified in the same way as the paths for output b . Therefore, the equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for weak bisimulation defined by equation A.45 and includes the initial states 0 and 0' of A_1 and A_2 , respectively. As a consequence, the PCA models A_1 and A_2 are weakly bisimilar.

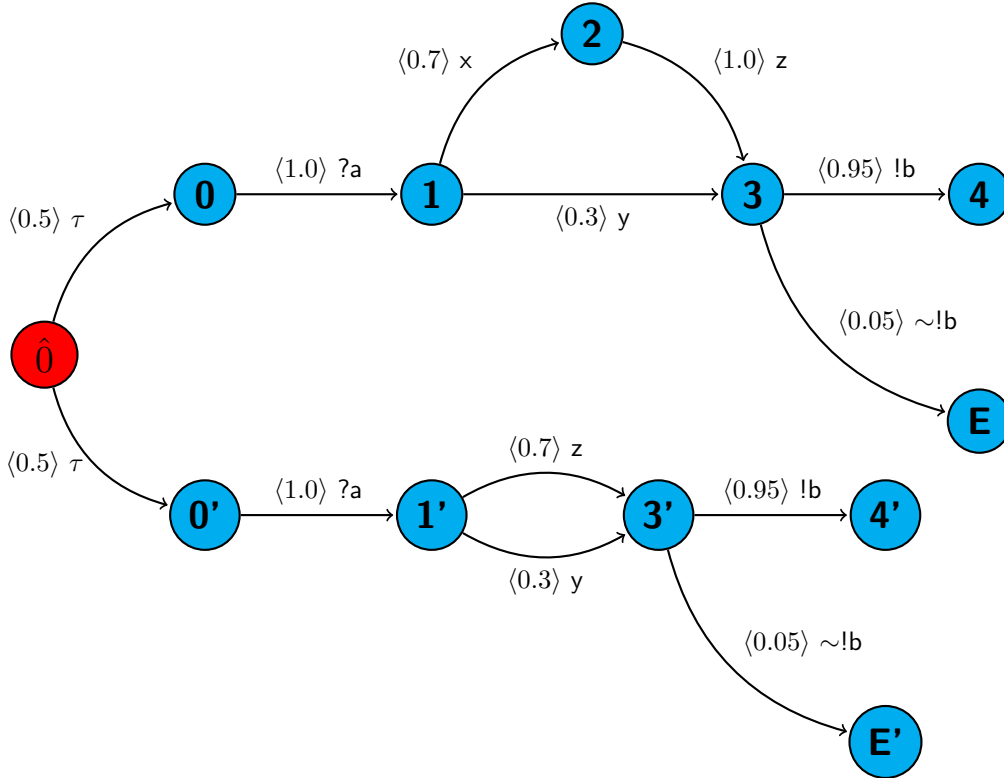


Figure A.10.: Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_1

A.4.3. Transformation T_2

In this Section we show that the reduced models produced by transformation T_2 and the original model are also weakly bisimilar. While transformation T_1 is applied to a single outgoing transition of a given state, transformation T_2 denotes the second main case of the reduction algorithm and considers multiple incoming transitions $(s_{src}, a_r, s_{dst}) \in \Delta_1$ to a given state s_{dst} labelled with actions $a_r \in \mathcal{E}_1^R$. When determining if these transitions are removed when constructing A_2 from A_1 , transformation T_2 considers the following two sub-cases:

- all the incoming transitions to state s_{dst} originate from the same source state s_{src} : $\rho_1^s(s_{dst}) = \{s_{src}\}$;
- the incoming transitions to state s_{dst} originate from different source states s_{src} .

In the latter case, the incoming transitions to state s_{dst} are grouped based on their source state and transformation T_2 is applied separately to each group of transitions.

Same source state s_{src}

The set of transitions (s_{src}, a_r, s_{dst}) from the same source state s_{src} are removed iff all the following conditions hold:

- the outgoing transitions of state s_{dst} are not labelled with input actions: $\Delta_1^e(s_{dst}) \cap \mathcal{E}_1^{in} = \emptyset$;
- either state s_{src} has outgoing transitions labelled with output actions or state s_{dst} , not both: $\Delta_e(s) \cap \mathcal{E}^{out} = \emptyset \vee \Delta_e(s') \cap \mathcal{E}^{out} = \emptyset$.

The first condition ensures that the aggregated probability $\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst})$ is only propagated forward to the successors transitions of state s_{dst} if these are not labeled with input actions in order to preserve their reactive semantics. The second condition guarantees that the set of outgoing transitions labelled with output actions is kept for each state. Note that this condition is necessary for preserving the compositional properties of the reduced model, as we show later in Section A.5. We describe in the next paragraphs how the PCA A_2 and the equivalence relationship $\mathcal{R}_{T_2}$ are defined by the transformation T_2 .

For all transitions $(s, a, s') \in \Delta_1$, except the transitions $\{(s_{src}, a_r, s_{dst})\}$, if the source state s is not the destination state s_{dst} of the deleted transition, the following steps are performed:

- state s is mapped to a state t and (s, t) is added to $\mathcal{R}_{T_2}$;
- state s' is mapped to a state t' and (s', t') is added to $\mathcal{R}_{T_2}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \mu_1(s, a, s') \quad (\text{A.99})$$

When case the transition source state s is the destination state s_{dst} the probability of the deleted transitions (s_{src}, a_r, s_{dst}) is propagated to the outgoing transitions of s_{dst} as follows:

- state s_{src} is mapped to a state t and (s_{src}, t) is added to $\mathcal{R}_{T_2}$;
- state s' is mapped to a state t' and (s', t') is added to $\mathcal{R}_{T_2}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \left(\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst}) \right) \cdot \mu_1(s_{dst}, a, s'). \quad (\text{A.100})$$

Note that rule A.100 is similar to the rule A.59 applied for forward propagation of a single transition, namely:

$$\mu_2(t, a, t') = \mu_1(s_{src}, a_r, s_{dst}) \cdot \mu_1(s_{dst}, a, s').$$

Moreover, the expression $\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst})$ can be replaced by $\mu_{1G}(s_{src}, a_r, s_{dst})$ using rule A.5 over a set of transitions labelled with internal actions in the set $\mathcal{E}_1^R \subseteq \mathcal{E}_1^{int}$. This case for transformation T_2 is analogous to the ones described for transformation T_1 . As a result, the same steps used for showing that the equivalence relationship $\mathcal{R}_{T_1}$ produced by transformation T_1 is a weak bisimulation can be applied to show that the equivalence relationship $\mathcal{R}_{T_2}$ produced by transformation T_2 is a weak bisimulation for the case where deleted transitions all come from the same source state s_{src} . For example, in rule A.80 and A.87 defined for T_1 , we substitute $\mu_1(s_{src}, a_r, s_{dst})$ by $\mu_{1G}(s_{src}, a_r, s_{dst})$ as follows:

$$\mu_{2G}(t, a_2^{int}, t_2) = \mu_{1G}(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_{dst}, a_2^{int}, s_2), \quad (\text{A.101})$$

$$\mu_{2G}(t_{i-1}, a_{i+1}^{int}, t_{i+1}) = \mu_{1G}(s_{src}, a_r, s_{dst}) \cdot \mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1}). \quad (\text{A.102})$$

For paths that do not include the delete transition(s), the equations A.74 and A.75 defined for the general expansion of $\mu_{1G}(s, \lambda, s')$ and $\mu_{2G}(t, \lambda, t')$ also apply to the case of transformation T_2 as rule A.58 is the same as A.99.

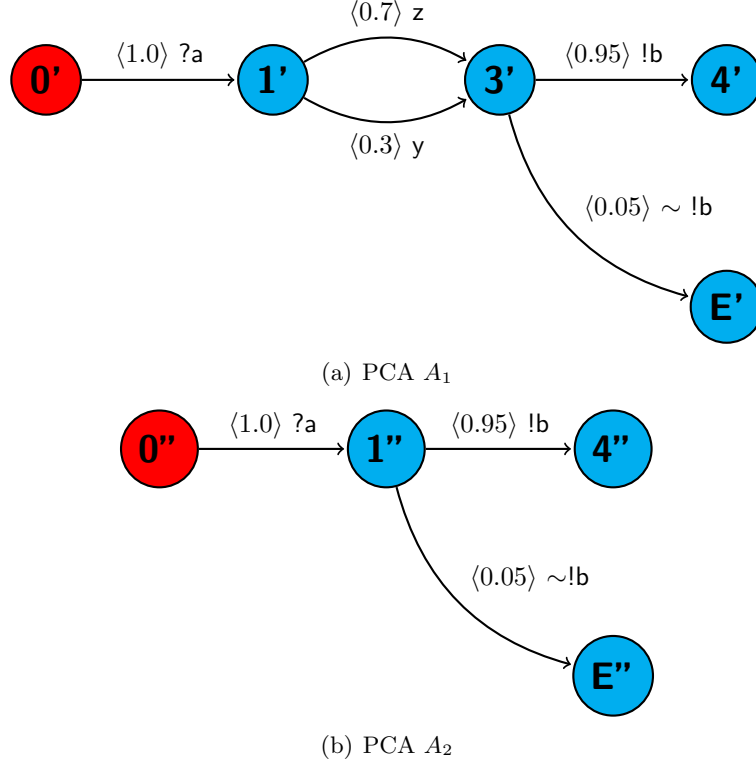


Figure A.11.: Examples of PCA Models for Weak Bisimulation - Transformation T_2 : Same Source State

We illustrate in the next paragraphs the application of transformation T_2 to the example PCA models A_1 and A_2 in Figure A.11 and their disjoint union $\hat{A}$ in Figure A.12. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and A_2 : $\{(0', 0''), (1', 1''), (4', 4''), (E', E'')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0', 0''\}, \{1', 1''\}, \{4', 4''\}, \{E', E''\}\}$. Using the weak bisimulation definition in equation A.45, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a weak bisimulation.

We omit the illustration for the case where $(s, t) = (0', 0'')$ and $C = \{1', 1''\}$ for the input action $?a$ as it has been verified in the previous example. We analyse instead the case where $(s, t) = (1', 1'')$ and $C = \{4', 4''\}$ for the output action $!b$:

$$\hat{\mu}_G(1', \tau^* !b \tau^*, \{4', 4''\}) = \hat{\mu}_G(1'', \tau^* !b \tau^*, \{4', 4''\}). \quad (\text{A.103})$$

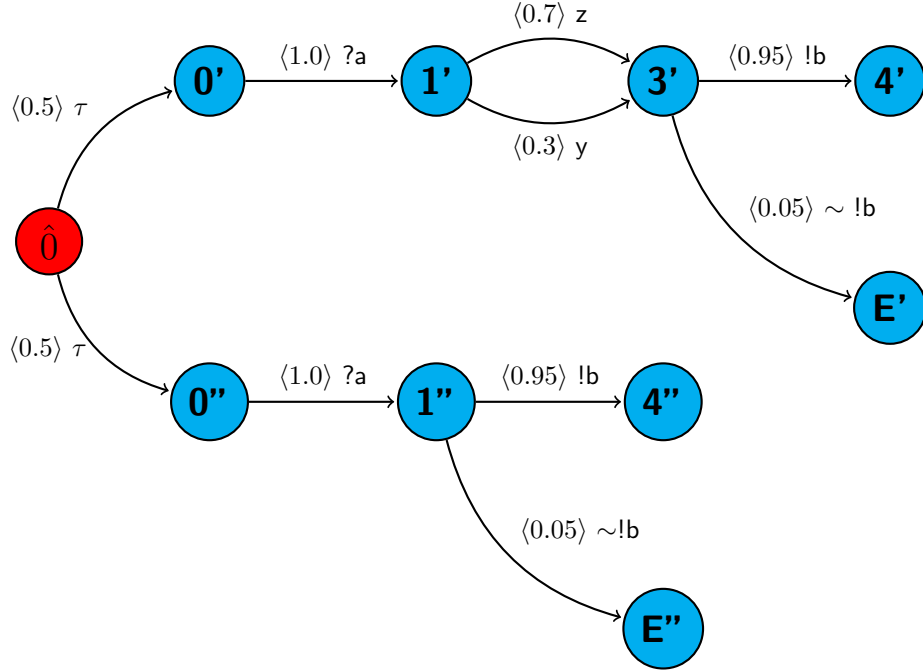


Figure A.12.: Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_2 : Same Source State

Given that the paths from state $1'$ to $4'$ and from state $1''$ to $4''$ do not include an internal transition after transitions labelled with output action $!b$, we simplify the previous conditions to

$$\hat{\mu}_G(1', \tau^* !b, \{4', 4''\}) = \hat{\mu}_G(1'', \tau^* !b, \{4', 4''\}). \quad (\text{A.104})$$

Using the expansion rule A.19 on $\hat{\mu}_G(1', \tau !b, \{4', 4''\})$ we obtain:

$$\hat{\mu}_G(1', \tau^*, 3') \cdot \hat{\mu}_G(3', !b, \{4', 4''\}) = \hat{\mu}_G(1'', \tau^*, 1'') \cdot \hat{\mu}_G(1'', !b, \{4', 4''\}). \quad (\text{A.105})$$

By replacing $\hat{\mu}_G$ by the corresponding cumulative probability functions from A_1 and A_2 and further expanding the sequences prefix τ^* we derive

$$\mu_{1G}(1', \tau, 3') \cdot \mu_{1G}(3', !b, \{4', 4''\}) = \mu_{2G}(1'', !b, \{4', 4''\}). \quad (\text{A.106})$$

and confirm that the condition A.103 for weak bisimulation is also verified for this case:

$$(0.7 + 0.3) \cdot 0.95 = 0.95 \quad \square. \quad (\text{A.107})$$

The verification of paths involving output action $\sim!b$ are verified in the same way as the paths for output b . Therefore, the equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for weak bisimulation defined by equation A.45 and includes the initial states 0 and $0'$ of A_1 and A_2 , respectively. As a consequence, the PCA models A_1 and A_2 are weakly bisimilar.

Multiple source states s_{src}

We now consider the case where the set of incoming transitions to state s_{dst} originate from different states. We describe the application of transformation T_2 to a set of transitions (s_{src}, a_r, s_{dst}) from a particular source state s_{src} , which is equivalently applied to the other group of transitions. The transitions (s_{src}, a_r, s_{dst}) are removed if one of the following conditions hold:

- the predecessor transitions of state s_{src} are not labelled with input actions and state s_{src} does not have other outgoing transitions to other states: $\rho_1^e(s_{src}) \cap \mathcal{E}_1^{in} = \emptyset \wedge \Delta_1^s(s_{src}) = \{s_{dst}\}$;
- the sum of the probability of all the transitions (s_{src}, a_r, s_{dst}) is equal to 1:

$$\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst}).$$

The first part of first condition ensures that the aggregated probability $\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst})$ is only propagated backward to the predecessor transitions of state s_{src} if these are not labeled with input actions in order to preserve their reactive semantics. If there are other transitions that can be chosen from state s_{src} , the second part guarantees the choice between those and the group of transitions (s_{src}, a_r, s_{dst}) is not broken. We describe in the next paragraphs how the PCA_2 and the equivalence relationship $\mathcal{R}_{T_2}$ are defined by the transformation T_2 .

For all transitions $(s, a, s') \in \Delta_1$, except the transitions $\{(s_{src}, a_r, s_{dst})\}$, if the source state s is not the destination state s_{dst} of the deleted transition, the following steps as performed:

- state s is mapped to a state t and (s, t) is added to $\mathcal{R}_{T_2}$;
- state s' is mapped to a state t' and (s', t') is added to $\mathcal{R}_{T_2}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \mu_1(s, a, s') \tag{A.108}$$

When the transition destination state s' is the destination state s_{src} , the probability of the deleted transitions (s_{src}, a_r, s_{dst}) is propagated to the predecessor transitions of s_{src} as follows:

- state s is mapped to a state t and (s, t) is added to $\mathcal{R}_{T_2}$;
- state s_{dst} is mapped to a state t' and (s_{dst}, t') is added to $\mathcal{R}_{T_2}$;
- transition (t, a, t') is added to Δ_2 and its probability is defined as

$$\mu_2(t, a, t') = \mu_1(s, a, s_{src}) \cdot \left(\sum_{(s_{src}, a_r, s_{dst}) \in \Delta_1} \mu_1(s_{src}, a_r, s_{dst}) \right). \quad (\text{A.109})$$

Similarly to the first case of transformation T_2 , the steps performed for paths that do not include the delete transitions are also applied to the second case of transformation T_2 as A.58 is the same as A.108.

We now consider the general case where the deleted transitions are between states s and s' and verify the following conditions

$$\mu_{1G}(s, \lambda, s') = \mu_{2G}(t, \lambda', t'), \text{ where } \lambda' = a_1^{int} \dots a_{i-1}^{int} a_{i+1}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}. \quad (\text{A.110})$$

We first apply the expansion rule A.19 to $\mu_{1G}(s, \lambda, s')$ and obtain

$$\begin{aligned} \mu_{1G}(s, \lambda, s') = & \sum_{s_{i-1} \in \mathcal{S}_1} \mu_{1G}(s, a_1^{int} \dots a_{i-1}^{int}, s_{i-1}) \cdot \mu_{1G}(s_{i-1}, a_i^{int}, s_i) \cdot \\ & \underbrace{\mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1})}_{\text{deleted transitions}} \cdot \mu_{1G}(s_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{l_1'}^{int} \dots b_{l_2}^{int}, s'), \end{aligned} \quad (\text{A.111})$$

where the deleted transitions $(s_{src}, a_r, s_{dst}) = (s_i, a_{i+1}^{int}, s_{i+1})$. Note that above equation only includes the summation $\sum_{s_{i-1} \in \mathcal{S}_1}$ as we are considering all sub-paths that finish at the source state s_{src} . We do not consider a summation over all incoming states s_i as the transformation T_2 is applied separately to the incoming transitions of state s_{i+1} based on their source state s_i . Additionally, in order for transitions $(s_i, a_{i+1}^{int}, s_{i+1})$ to be deleted, all the transitions from state s_i have s_{i+1} as the destination state, *i.e.* the state s_i does not contain other outgoing transitions apart from the deleted transitions. As a result, according to rule A.109, the aggregated probability of deleted transitions $\mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1})$ is propagated backwards to transitions the last transition $(s_{i-1}, a_i^{int}, s_i)$ of all the incoming sub-paths to the source state s_i of deleted transitions.

Furthermore, by applying the same expansion to $\mu_{2G}(t, \lambda', t')$ we obtain

$$\begin{aligned} \mu_{2G}(t, \lambda, t') = \sum_{t_{i-1} \in \mathcal{S}_2} \mu_{2G}(t, a_1^{int} \dots a_{i-1}^{int}, t_{i-1}) \cdot \mu_{2G}(t_{i-1}, a_i^{int}, t_{i+1}) \cdot \\ \mu_{2G}(t_{i+1}, a_{i+2}^{int} \dots a_{l_1}^{int} a b_{1'}^{int} \dots b_{l_2}^{int}, t'). \end{aligned} \quad (\text{A.112})$$

According to rule A.109 the probability of $\mu_{2G}(t_{i-1}, a_i^{int}, t_{i+1})$ is defined as

$$\mu_{2G}(t_{i-1}, a_i^{int}, t_{i+1}) = \mu_{1G}(s_{i-1}, a_i^{int}, s_i) \cdot \mu_{1G}(s_i, a_{i+1}^{int}, s_{i+1}). \quad (\text{A.113})$$

The same steps can be directly applied to the incoming transitions to state s_{i+1} of other source states s_i . Similarly, the case where the deleted transition $(s_{src}, a_r, s_{dst}) = (s_i, a_{i+1}^{int}, s_{i+1})$ are the last transitions in the first path that precedes the visible action a in the sequence $\lambda = a_1^{int} \dots a_{l_1-1}^{int} a_{l_1}^{int} a b_{1'}^{int} \dots b_{l_2}^{int}$ is also equivalent to the previous case as the visible action needs to be an output action for the transition s_{src}, a_r, s_{dst} and its probability to be propagated. We have covered the two cases of transformation T_2 and therefore shown that it produces a weakly bisimilar model as defined by the equivalence relationship $\mathcal{R}_{T_2}$.

Example

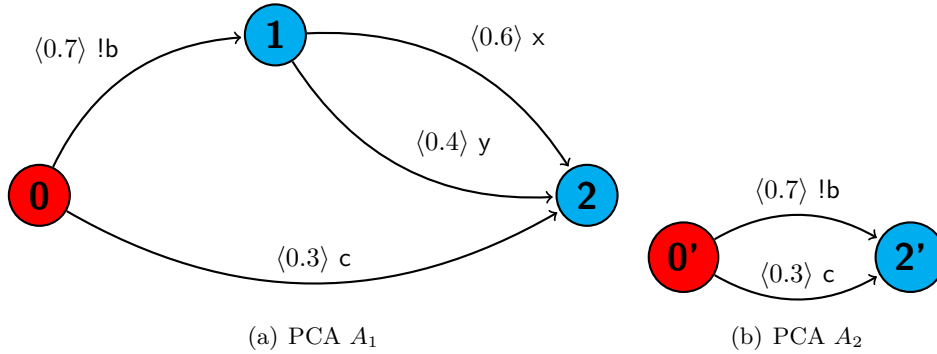


Figure A.13.: Examples of PCA Models for Weak Bisimulation - Transformation T_2 : Multiple Source States

We illustrate in the next paragraphs the application of the above steps to the example PCA models A_1 and A_2 in Figure A.7 and their disjoint union $\hat{A}$ in Figure A.14. We specify the equivalence relationship $\mathcal{R}_{12}$ between A_1 and A_2 : $\{(0, 0'), (2, 2')\}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{\{0, 0'\}, \{2, 2'\}\}$. Using the weak bisimulation definition in equation A.45, we verify below the equivalence relationship $\mathcal{R}_{12}$ is a weak bisimulation.

We start by considering the case where $(s, t) = (0, 0')$ and $C = \{2, 2'\}$ for the output action $!b$:

$$\hat{\mu}_G(0, \tau^* !b \tau^*, \{2, 2'\}) = \hat{\mu}_G(0', \tau^* !b \tau^*, \{2, 2'\}). \quad (\text{A.114})$$

By applying the expansion rules A.7, A.17 and A.19 we obtain

$$\mu_{1G}(0, !b, 1) \cdot \mu_{1G}(1, \tau, 2) = \mu_{2G}(0', !b, 2'). \quad (\text{A.115})$$

By replacing μ_{1G} and μ_{2G} by the probabilities of the corresponding transitions we verify the condition A.115:

$$0.7 \cdot (0.6 + 0.4) = 0.7 \quad \square. \quad (\text{A.116})$$

We now consider the case where $(s, t) = (0, 0')$ and $C = \{2, 2'\}$ for paths that only include internal actions:

$$\hat{\mu}_G(0, \tau^*, \{2, 2'\}) = \hat{\mu}_G(0', \tau^*, \{2, 2'\}). \quad (\text{A.117})$$

By applying the expansion rules A.7, A.17 and A.19 we obtain

$$\mu_{1G}(0, \tau, 2) = \mu_{2G}(0', \tau, 2'). \quad (\text{A.118})$$

By replacing μ_{1G} and μ_{2G} by the probabilities of the corresponding transitions we verify the condition A.118:

$$0.3 = 0.3 \quad \square. \quad (\text{A.119})$$

Therefore, the equivalence relationship $\mathcal{R}_{12}$ verifies the conditions for weak bisimulation defined by equation A.45 and includes the initial states 0 and $0'$ of A_1 and A_2 , respectively. As a consequence, the PCA models A_1 and A_2 are weakly bisimilar.

Remarks

Note that the probabilities of other paths is not changed by the transformations T_1 and T_2 when applied to transitions (s_{src}, a_r, s_{dst}) as forward propagation is only performed **only** when there

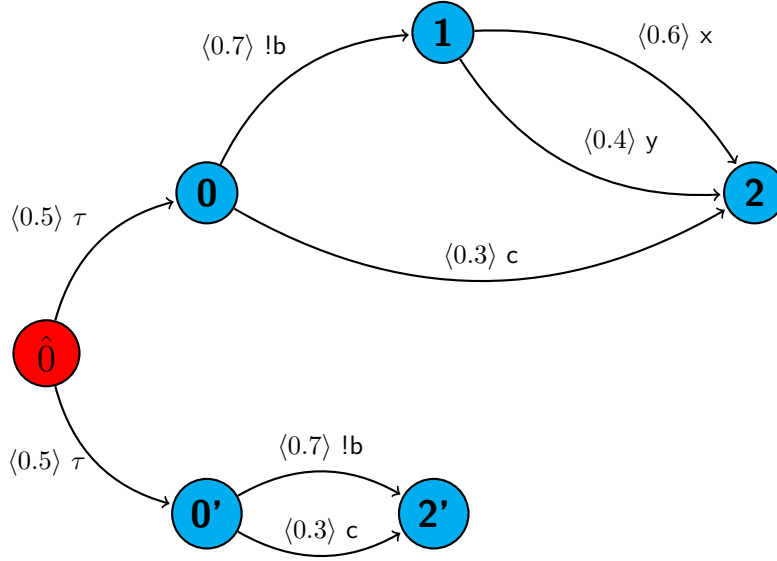


Figure A.14.: Disjoint Union PCA $\hat{A}$ for Weak Bisimulation Between PCA Models A_1 and A_2 - Transformation T_1

are no other incoming paths to state s_{src} and backward propagation is applied **only** when the state s_{src} does not have other outgoing transitions.

A.4.4. Transitivity

In the previous Section we have shown that the automaton produced by the two transformations used by the reduction algorithm is weakly bisimilar to the original automaton. In this Section we show that the equivalence relationships $\mathcal{R}_{T_1}$ and $\mathcal{R}_{T_2}$ corresponding to the two transformations are transitive.

Given $A_1 = \langle \mathcal{S}_1, q_1, \mathcal{E}_1, \Delta_1, \mu_1 \rangle$, $A_2 = \langle \mathcal{S}_2, q_2, \mathcal{E}_2, \Delta_2, \mu_2 \rangle$ and $A_3 = \langle \mathcal{S}_3, q_3, \mathcal{E}_3, \Delta_3, \mu_3 \rangle$, consider $A_1 \approx A_2$ and $A_2 \approx A_3$ as defined by the equivalence relationships $\mathcal{R}_{12}$ and $\mathcal{R}_{23}$, respectively. We want to show that $A_1 \approx A_2 \wedge A_2 \approx A_3 \implies A_1 \approx A_3$ through an equivalence relationship $\mathcal{R}_{13}$.

According to the weak bisimulation condition for PCA, A_1 is weakly bisimilar to A_2 through the equivalence relationship $\mathcal{R}_{12}$ on the disjoint union $\hat{A}_{12}$ of A_1 and A_2 iff $(q_1, q_2) \in \mathcal{R}_{12}$ and $\forall (s_{12}, t_{12}) \in \mathcal{R}_{12}, \forall a \in \hat{\mathcal{E}}_{12}, \forall C_{12} \in \hat{\mathcal{S}}_{C_{12}}$:

$$\hat{\mu}_{G12}(s_{12}, \tau^* a \tau^*, C_{12}) = \hat{\mu}_{G12}(t_{12}, \tau^* a \tau^*, C_{12}) \quad (\text{A.120})$$

$$\hat{\mu}_{G12}(s_{12}, \tau^* a \tau^*, \{s'_{12}, t'_{12}\}) = \hat{\mu}_{G12}(t_{12}, \tau^* a \tau^*, \{s'_{12}, t'_{12}\}) \quad (\text{A.121})$$

Similarly for A_2 and A_3 , the equivalence relationship $\mathcal{R}_{23}$ needs to hold the following properties on the disjoint union $\hat{A}_{23}$ of A_2 and A_3 , $(q_2, q_3) \in \mathcal{R}_{23}$ and $\forall (s_{23}, t_{23}) \in \mathcal{R}_{23}, \forall a \in \hat{\mathcal{E}}_{23}, \forall C_{23} \in \hat{\mathcal{S}}_{C_{23}}$:

$$\hat{\mu}_{G23}(s_{23}, \tau^* a \tau^*, C_{23}) = \hat{\mu}_{G23}(t_{23}, \tau^* a \tau^*, C_{23}) \quad (\text{A.122})$$

$$\hat{\mu}_{G23}(s_{23}, \tau^* a \tau^*, \{s'_{23}, t'_{23}\}) = \hat{\mu}_{G23}(t_{23}, \tau^* a \tau^*, \{s'_{23}, t'_{23}\}) \quad (\text{A.123})$$

In order for A_1 to be weakly bisimilar to A_3 , an equivalence relationship $\mathcal{R}_{13}$ on the disjoint union $\hat{A}_{13}$ of A_1 and A_3 needs to hold the following properties $(q_1, q_3) \in \mathcal{R}_{13}$ and $\forall (s_{13}, t_{13}) \in \mathcal{R}_{13}, \forall a \in \hat{\mathcal{E}}_{13}, \forall C_{13} \in \hat{\mathcal{S}}_{C_{13}}$:

$$\hat{\mu}_{G13}(s_{13}, \tau^* a \tau^*, C_{13}) = \hat{\mu}_{G13}(t_{13}, \tau^* a \tau^*, C_{13}) \quad (\text{A.124})$$

$$\hat{\mu}_{G13}(s_{13}, \tau^* a \tau^*, \{s'_{13}, t'_{13}\}) = \hat{\mu}_{G13}(t_{13}, \tau^* a \tau^*, \{s'_{13}, t'_{13}\}) \quad (\text{A.125})$$

Consider that the equivalence relationship $\mathcal{R}_{13}$ is defined as follows based upon the equivalence relationships $\mathcal{R}_{12}$ and $\mathcal{R}_{23}$:

$$\begin{aligned} \forall (s_{12}, t_{12}) \in \mathcal{R}_{12} \wedge (s_{23}, t_{23}) \in \mathcal{R}_{23} \\ \text{Assuming } s_{12} \in \mathcal{S}_1 \implies t_{12} \in \mathcal{S}_2 \\ \text{Assuming } s_{23} \in \mathcal{S}_2 \implies t_{23} \in \mathcal{S}_3 \\ \text{If } t_{12} = s_{23} \end{aligned} \quad (\text{A.126})$$

$$(s_{12}, t_{23}) \in \mathcal{R}_{13} \wedge [s_{12}]_{\mathcal{R}_{13}} = \{t_{13}\} \wedge [t_{23}]_{\mathcal{R}_{13}} = \{s_{12}\} \quad (\text{A.127})$$

We demonstrate that A_1 is weakly bisimilar to A_3 as defined by the equivalence relationship $\mathcal{R}_{13}$.

Consider $s_{12} = s_1 \in \mathcal{S}_1$ and $s'_{12} = s'_1 \in \mathcal{S}_1 \implies t_{12} = s_2 \in \mathcal{S}_2$ and $t'_{12} = s'_2 \in \mathcal{S}_2$, then

$$\hat{\mu}_{G13}(s_{13}, \tau^* a \tau^*, \{s'_{13}, t'_{13}\}) = \mu_{G1}(s_1, \tau^* a \tau^*, s'_1) \quad (\text{A.128})$$

and

$$\hat{\mu}_{G13}(t_{13}, \tau^* a \tau^*, \{s'_{13}, t'_{13}\}) = \mu_{G2}(s_2, \tau^* a \tau^*, s'_2) \quad (\text{A.129})$$

Equivalently for A_2 and A_3 , consider $s_{23} = s_2 \in \mathcal{S}_2$ and $s'_{23} = s'_2 \in \mathcal{S}_2 \implies t_{23} = s_3 \in \mathcal{S}_3$ and $t'_{23} = s'_3 \in \mathcal{S}_3$, then

$$\hat{\mu}_{G23}(s_{23}, \tau^* a \tau^*, \{s'_{23}, t'_{23}\}) = \mu_{G2}(s_2, \tau^* a \tau^*, s'_2) \quad (\text{A.130})$$

and

$$\hat{\mu}_{G23}(t_{23}, \tau^* a \tau^*, \{s'_{23}, t'_{23}\}) = \mu_{G3}^{F_3}(s_3, \tau^* a \tau^*, s'_3) \quad (\text{A.131})$$

From $A_1 \approx A_2$ it holds that

$$\mu_{G1}(s_1, \tau^* a \tau^*, s'_1) = \mu_{G2}(s_2, \tau^* a \tau^*, s'_2),$$

and from $A_2 \approx A_3$ it holds that

$$\mu_{G2}(s_2, \tau^* a \tau^*, s'_2) = \mu_{G3}^{F_3}(s_3, \tau^* a \tau^*, s'_3).$$

According to rule A.127 for defining the equivalence relationship $(s_1, s_3) \in \mathcal{R}_{13}$ if $(s_1, s_2) \in \mathcal{R}_{12}$ and $(s_2, s_3) \in \mathcal{R}_{23}$. Consequently it holds that

$$\mu_{G1}(s_1, \tau^* a \tau^*, s'_1) = \mu_{G2}(s_2, \tau^* a \tau^*, s'_2) = \mu_{G3}^{F_3}(s_3, \tau^* a \tau^*, s'_3) \implies$$

$$\hat{\mu}_{G13}(s_{13}, \tau^* a \tau^*, s'_{13}) = \hat{\mu}_{G13}(t_{13}, \tau^* a \tau^*, t'_{13}).$$

The same steps can be applied to the condition for weak bisimulation for paths that only include internal actions.

Example

Using PCA models A_1 , A_2 and A_3 in Figure A.15 we now describe the application of the previous rules to define a transitive relationship between equivalence relationships. Consider the model A_2 (Figure A.15(b)) has been obtained by applying the transformation T_1 to A_1 (Figure A.15(a)) w.r.t. transition $1 \xrightarrow{x, 0.7} 2$. The corresponding equivalence relationship $\mathcal{R}_{12}$ is defined as $\mathcal{R}_{12} = \{(0, 0'), (1, 1'), (3, 3'), (4, 4'), (E, E')\}$. On the other hand, the model A_3 (Figure A.15(c)) has been obtained by applying transformation T_2 to model A_2 w.r.t. to transitions $1' \xrightarrow{z, 0.7} 3'$ and $1' \xrightarrow{y, 0.3} 3'$ and the corresponding equivalence relationship $\mathcal{R}_{23}$ is defined as $\mathcal{R}_{23} = \{(0', 0''), (1', 1''), (4', 4''), (E', E'')\}$. We now describe how an equivalence relationship

$\mathcal{R}_{13}$ is defined from $\mathcal{R}_{12}$ and $\mathcal{R}_{23}$ and show that models A_1 and A_3 are weakly bisimilar given that $A_1 \approx A_2$ and $A_2 \approx A_3$.

From rules A.126 and A.126 we obtain the following equivalence relationship $\mathcal{R}_{13} = \{ (0, 0''), (1, 1''), (4, 4''), (E, E'') \}$. The corresponding set of equivalence classes $\hat{\mathcal{S}}_C$ is defined as $\{ \{0, 0''\}, \{1, 1''\}, \{4, 4''\}, \{E, E''\} \}$. Using the weak bisimulation definition in equation A.45, we verify below the equivalence relationship $\mathcal{R}_{13}$ is a weak bisimulation.

We start by considering the case where $(s, t) = (0, 0'')$ and $C = \{1, 1''\}$ for the input action a :

$$\hat{\mu}_G(0, \tau^* ?a \tau^*, \{1, 1''\}) = \hat{\mu}_G(0'', \tau^* ?a \tau^*, \{1, 1''\}). \quad (\text{A.132})$$

Given that there is a single path between states 0 and 1 as well as between states $0''$ and $1''$ we derive the following from the expansion rule A.19:

$$\hat{\mu}_G(0, \epsilon, 0) \cdot \hat{\mu}_G(0, ?a, \{1, 1''\}) = \hat{\mu}_G(0'', \epsilon, 0'') \cdot \hat{\mu}_G(0'', \tau^* ?a \tau^*, \{1, 1''\}). \quad (\text{A.133})$$

By applying rule A.17 to $\hat{\mu}_G(0, \epsilon, 0)$ and $\hat{\mu}_G(0'', \epsilon, 0'')$ as well as the expansion rule A.7 when μ_G is applied to a set of states, we obtain

$$1 \cdot (\hat{\mu}_G(0, ?a, 1) + \hat{\mu}_G(0, ?a, 1'')) = 1 \cdot (\hat{\mu}_G(0'', a, 1) + \hat{\mu}_G(0'', a, 1'')). \quad (\text{A.134})$$

Given that the transitions $\hat{\mu}_G(0, ?a, 1'')$ and $\hat{\mu}_G(0'', ?a, 1)$ do not exist in $\hat{\Delta}$, we derive

$$\hat{\mu}_G(0, ?a, 1) + 0 = 0 + \hat{\mu}_G(0'', ?a, 1''). \quad (\text{A.135})$$

By replacing $\hat{\mu}_G(0, a, 1)$ by $\mu_{1G}(0, ?a, 1)$ and $\hat{\mu}_G(0'', ?a, 1'')$ by $\mu_{3G}(0'', ?a, 1'')$ we confirm that the condition for weak bisimulation is verified for this case:

$$1 = 1 \quad \square \quad (\text{A.136})$$

Consider now the case where $(s, t) = (1, 1'')$ and $C = \{3, 3''\}$ for the output action $!b$:

$$\hat{\mu}_G(1, \tau^* !b \tau^*, \{3, 3''\}) = \hat{\mu}_G(1'', \tau^* !b \tau^*, \{3, 3''\}). \quad (\text{A.137})$$

While there is only one transition in the path between states $1''$ and $3''$ in $\hat{A}$, the paths between states 1 and 3 include transitions through intermediate state 2. Therefore, the general condition A.21 for weak bisimulation is mapped to:

$$\hat{\mu}_G(1, \tau !b, \{3, 3''\}) = \hat{\mu}_G(1'', !b, \{3, 3''\}). \quad (\text{A.138})$$

Using the expansion rule A.19 on $\hat{\mu}_G(1, \tau !b, \{3, 3''\})$ we obtain:

$$\sum_{s \in \hat{S}} \hat{\mu}_G(1, \tau, s) \cdot \hat{\mu}_G(s, !b, \{3, 3''\}) = \hat{\mu}_G(1'', !b, \{3, 3''\}). \quad (\text{A.139})$$

We obtain the following by also applying the rule A.7 when μ_G is applied to a set of states:

$$\hat{\mu}_G(1, \tau, 2) \cdot \left(\hat{\mu}_G(2, !b, 3) + \hat{\mu}_G(2, !b, 3'') \right) = \hat{\mu}_G(1'', !b, 3) + \hat{\mu}_G(1'', !b, 3''). \quad (\text{A.140})$$

By replacing $\hat{\mu}_G$ by the corresponding cumulative probability functions from A_1 and A_3 we derive

$$\mu_{1G}(1, \tau, 2) \cdot \left(\mu_{1G}(2, !b, 3) + 0 \right) = 0 + \mu_{3G}(1'', !b, 3''). \quad (\text{A.141})$$

and confirm that the condition for weak bisimulation is also verified for this case:

$$(0.7 + 0.3) \cdot 0.95 = 0.95 \quad \square \quad (\text{A.142})$$

Finally, consider the case where $(s, t) = (1, 1'')$ and $C = \{E, E''\}$ for the output failure action $\sim !b$:

$$\hat{\mu}_G(1, \tau^* \sim !b \tau^*, \{E, E''\}) = \hat{\mu}_G(1'', \tau^* \sim !b \tau^*, \{E, E''\}). \quad (\text{A.143})$$

By applying the same steps as for the previous case, we obtain

$$\mu_{1G}(1, \tau, 2) \cdot \left(\mu_{1G}(2, \sim !b, E) + 0 \right) = 0 + \mu_{3G}(1'', \sim !b, E''). \quad (\text{A.144})$$

and confirm that the condition for weak bisimulation is also verified for this case:

$$(0.7 + 0.3) \cdot 0.05 = 0.05 \quad \square \quad (\text{A.145})$$

The equivalence relationship $\mathcal{R}_{13}$ verifies the conditions for weak bisimulation defined by equation A.45 and includes the initial states 0 and $0''$ of A_1 and A_3 , respectively. As a consequence, the PCA models A_1 and A_3 are weakly bisimilar.

We have shown that the equivalence relationships associated with each transformation are weakly bisimilar and transitive. In the general case, a sequence of n transformations produces a weakly bisimilar model A_n from an original model A_1 as defined by the equivalence relationship $\mathcal{R}_{1n}$. Therefore, the reduced model produced the hiding operator is a weakly bisimilar model as the reduction algorithm applies a single transformation to the incoming transitions of each state.

A.5. Congruence

In the previous sections we demonstrated that the reduced models produced by the reduction algorithm preserve the reachability properties of the original model by showing that the equivalence relationships produced by each reduction case are weakly bisimilar, and the application of a sequence of transitions also produces a weakly bisimilar model. In this Section we show that these relationships are also a congruence on PCA. In other words, if two PCA models A_1 and A_2 are weakly bisimilar as defined by an equivalence relationship $\mathcal{R}_{T_i}$ ($A_1 \mathcal{R}_{T_i} A_2$), the models resulting from the application of any operator on PCA models to A_1 and/or A_2 are also weakly bisimilar. Formally, if $A_1 \approx A_2$ then:

$$A_1 \setminus \mathcal{E}^{nobs} \approx A_2 \setminus \mathcal{E}^{nobs} \text{ (hiding)} \quad (\text{A.146})$$

$$A_1 @ \mathcal{E}^{unused} \approx A_2 @ \mathcal{E}^{unused} \text{ (interface)} \quad (\text{A.147})$$

$$A_1 / L \approx A_2 / L \text{ (re-labelling)} \quad (\text{A.148})$$

$$A_1 \parallel B \approx A_2 \parallel B \text{ (parallel composition)} \quad (\text{A.149})$$

Consequently, given an original model A_1 and an expression involving any operator supported by PCA, the model A_1 can be replaced by a weakly bisimilar model A_2 whilst preserving weak bisimulation. In the next Sections we show this result for each of the listed operators.

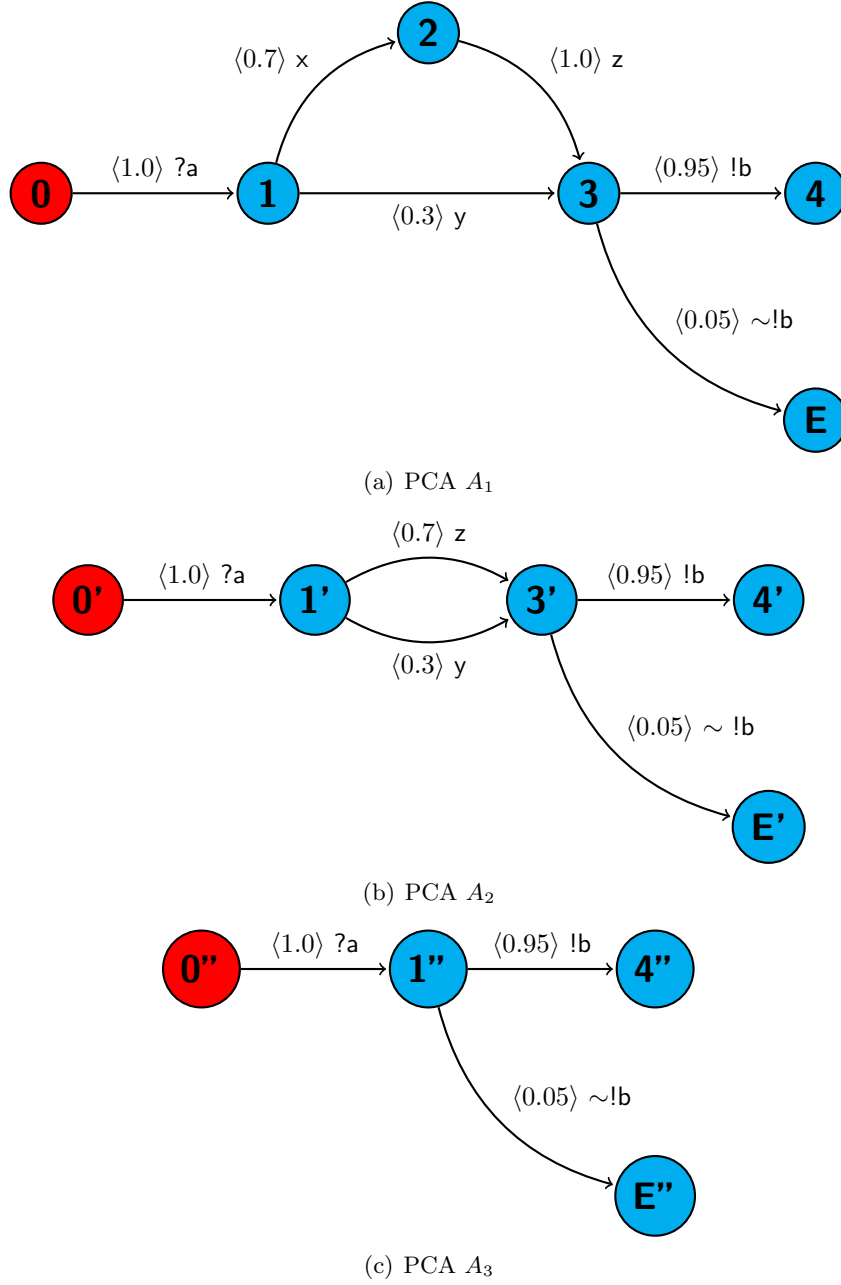
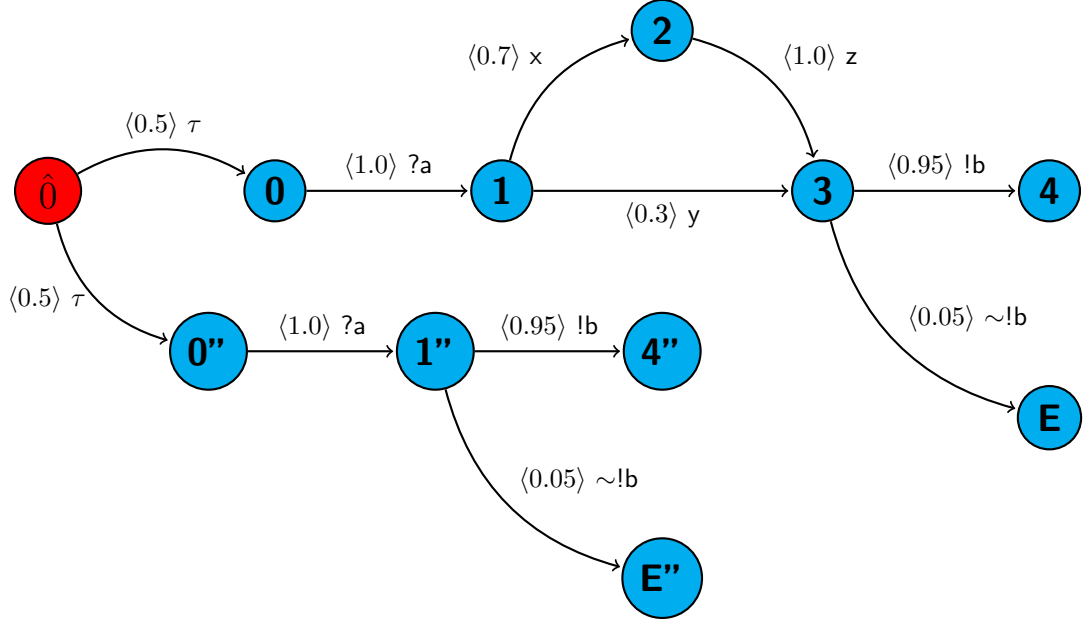


Figure A.15.: Examples of PCA Models for Transitivity Property

Figure A.16.: Disjoint Union PCA $\hat{A}$ for Transitivity Property

A.5.1. Hiding

$$\text{Given } A_1 \approx A_1 \setminus \mathcal{E}^{nobs} \quad (\text{A.150})$$

$$\text{and } A_2 \approx A_2 \setminus \mathcal{E}^{nobs} \quad (\text{A.151})$$

$$\text{then } A_1 \setminus \mathcal{E}^{nobs} \approx A_2 \implies A_1 \setminus \mathcal{E}^{nobs} \approx A_2 \setminus \mathcal{E}^{nobs} \quad (\text{A.152})$$

The conditions A.150, A.151 and A.152 are verified using the transitivity property associated with the transformations T_i .

A.5.2. Interface

Given two models $A_1 \approx A_2$ and the corresponding equivalence relationship $\mathcal{R}$ that verifies the conditions for weak bisimulation on their disjoint union $\hat{A}$. We show in the next paragraphs how the same equivalence relationship holds when applying the interface operator $@$ w.r.t. set of input actions $\mathcal{E}_{unbound}^{in}$ to both A_1 and A_2 . Consider $A'_1 = A_1 @ \mathcal{E}_{unbound}^{in}$ and $A'_2 = A_2 @ \mathcal{E}_{unbound}^{in}$ and their disjoint union $\hat{A}'$.

In the following paragraphs we show that $\mathcal{R}$ is also a weak bisimulation on $\hat{A}'$, such that $\forall (s, t) \in \mathcal{R}, \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \mathcal{E}^{visible}$ the following conditions hold:

$$\hat{\mu}_G(s, \tau^* a \tau^*, C) = \hat{\mu}_G(t, \tau^* a \tau^*, C), \quad (\text{A.153})$$

$$\hat{\mu}_G(s, \tau^*, C) = \hat{\mu}_G(t, \tau^*, C) \text{ where } \tau \in \mathcal{E}_1^{int}. \quad (\text{A.154})$$

If in equation A.153 the visible action a corresponds to an output action or an input action not in $\mathcal{E}_{unbound}^{in}$, then the condition is verified given that $A_1 \approx A_2$. In the case that action a is an input action in $\mathcal{E}_{unbound}^{in}$ and assuming $s, s' \in \mathcal{S}_1$ and $t, t' \in \mathcal{S}_2$, we expand the equation A.153 as follows:

$$\sum_{s'' \in \mathcal{S}_1} \mu_{1G}(s, \tau^*, s'') \cdot \mu_{1G}(s'', a \tau^*, s') = \sum_{t'' \in \mathcal{S}_2} \mu_{2G}(t, \tau^*, t'') \cdot \mu_{2G}(t'', a \tau^*, t') \quad (\text{A.155})$$

From the constraints of transformations T_1 and T_2 where transitions labelled with internal actions cannot be propagated to transitions labelled with input actions, we obtain that states s'' and t'' are weakly bisimilar $((s'', t'') \in \mathcal{R})$ thus $\sum_{s'' \in \mathcal{S}_1} \mu_{1G}(s, \tau^*, s'') = \sum_{t'' \in \mathcal{S}_2} \mu_{2G}(t, \tau^*, t'')$ which verifies condition A.154.

A.5.3. Relabelling

Given two models $A_1 \approx A_2$ and the corresponding equivalence relationship $\mathcal{R}$ that verifies the conditions for weak bisimulation on their disjoint union $\hat{A}$. We show in the next paragraphs how the same equivalence relationship holds when applying the relabelling operator w.r.t. relation L to both A_1 and A_2 . Consider $A'_1 = A_1 L$ and $A'_2 = A_2 / L$ and their disjoint union $\hat{A}'$.

When the relation is $\{a\} \times L$, $a \in \mathcal{E}_1$, each transition $(s, a, s') \in \Delta_1$ is replaced by $\#(L)$ transitions labelled with the action labels in L and the probability associated with those transitions is re-defined to $\mu'_1(s, a, s') = \frac{\mu_1(s, a, s')}{\#(L)}$. Note that the original probability $\mu_1(s, a, s')$ can be defined as

$$\mu_1(s, a, s') = \sum_{(s, a, s') \in \Delta_1} \mu'_1(s, a, s')$$

which is the same as definition μ_G in equation A.5. The same steps can be applied to A_2 to derive the new probabilities of re-labelled transitions. Therefore, the $\mathcal{R}$ is also a weak bisimulation on $\hat{A}'$, such that $\forall (s, t) \in \mathcal{R}$, $\forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \mathcal{E}^{visible}$ the following conditions hold:

$$\hat{\mu}_G(s, \tau^* a \tau^*, C) = \hat{\mu}_G(t, \tau^* a \tau^*, C), \quad (\text{A.156})$$

$$\hat{\mu}_G(s, \tau^*, C) = \hat{\mu}_G(t, \tau^*, C) \text{ where } \tau \in \mathcal{E}_1^{int}. \quad (\text{A.157})$$

A.5.4. Parallel Composition

Consider two models $A_1 \approx A_2$ and the corresponding equivalence relationship $\mathcal{R}$ which verifies the conditions for weak bisimulation on their disjoint union $\hat{A}$. Given another B , we define the composite models $A_1 B = A_1 \parallel B = \langle \mathcal{S}_{A_1 B}, q_{A_1 B}, \mathcal{E}_{A_1 B}, \Delta_{A_1 B}, \mu_{A_1 B} \rangle$ and $A_2 B = A_2 \parallel B = \langle \mathcal{S}_{A_2 B}, q_{A_2 B}, \mathcal{E}_{A_2 B}, \Delta_{A_2 B}, \mu_{A_2 B} \rangle$. We define an equivalence relationship $\mathcal{R}'$ on $\hat{A}_{\parallel}$ the disjoint union of $A_1 B$ and $A_2 B$ as follows:

$$\forall \langle s_{A_2}, s_B \rangle \in \mathcal{S}_{A_2 B} \wedge (s_{A_1}, s_{A_2}) \in \mathcal{R}, (\langle s_{A_1}, s_B \rangle, \langle s_{A_2}, s_B \rangle) \in \mathcal{R}' \text{ iff} \quad (\text{A.158})$$

$$\Delta_{A_2 B}(\langle s_{A_2 B}, s_B \rangle) = \emptyset \vee$$

$$(\rho_{A_2}^e(s_{A_2}) \cap \rho_B^e(s_B)) \neq \emptyset \quad (A_2 \text{ and } B \text{ make a synchronous move})$$

The equivalence relationship $\mathcal{R}'$ defines as bisimilar the states where both A_2 and B make a synchronous move as well as deadlock states. If A_2 synchronises with B by executing an output action a that leads to a state $s_{A_2} \in \mathcal{S}_{A_2}$, then A_1 also makes a synchronous move with B through action a . When constructing A_2 by reducing A_1 , if no transition is deleted immediately prior or after to the transition labelled with output action a leading to state $s_{A_1} \in \mathcal{S}_{A_1}$, then the pair of bisimilar states (s_{A_1}, s_{A_2}) is added to $\mathcal{R}'$. The other states represent intermediate states where the models execute independently their non-synchronisable actions.

In the following paragraphs we show that $\mathcal{R}'$ is a weak bisimulation on $\hat{A}_{\parallel}$, such that $\forall (s, t) \in \mathcal{R}'$, $\forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \text{shared}(A_1, B) = \mathcal{E}_{A_1} \cap \mathcal{E}_B$ the following conditions hold:

$$\hat{\mu}_G(s, \tau^* a \tau^*, C) = \hat{\mu}_G(t, \tau^* a \tau^*, C), \quad (\text{A.159})$$

$$\hat{\mu}_G(s, \tau^*, C) = \hat{\mu}_G(t, \tau^*, C) \text{ where } \tau \in (\mathcal{E}_{A_1} \cup \mathcal{E}_B) \setminus \text{shared}(A_1, B). \quad (\text{A.160})$$

Note that $\text{shared}(A_1, B)$ only contains synchronisable interface actions as we assume the models A_1 and B are compatible for parallel composition. The special action label τ refers in this case

to all actions on which the models do not synchronise, which include by definition their internal actions but also input and output actions which are not shared between the two models.

Given how the equivalence relationship $\mathcal{R}'$ is constructed, and assuming $s = \langle s_{A_1}, s_B \rangle$, $s' = \langle s'_{A_1}, s'_B \rangle$, $t = \langle s_{A_2}, s_B \rangle$ and $t' = \langle s'_{A_2}, s'_B \rangle$, we redefine the condition A.159 as follows:

$$\mu_{A_1 B-G}(\langle s_{A_1}, s_B \rangle, \tau^* a \tau^*, \langle s'_{A_1}, s'_B \rangle) = \mu_{A_2 B-G}(\langle s_{A_2}, s_B \rangle, \tau^* a \tau^*, \langle s'_{A_2}, s'_B \rangle). \quad (\text{A.161})$$

By applying the expansion rule A.19 for a sequence of actions to $\mu_{A_1 B-G}(\langle s_{A_1}, s_B \rangle, \tau^* a \tau^*, \langle s'_{A_1}, s'_B \rangle)$ we obtain:

$$\begin{aligned} & \sum_{\langle s''_{A_1}, s''_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1 B-G}(\langle s_{A_1}, s_B \rangle, \tau^*, \langle s''_{A_1}, s''_B \rangle) \cdot \\ & \left(\sum_{\langle s'''_{A_1}, s'''_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1 B-G}(\langle s''_{A_1}, s''_B \rangle, a, \langle s'''_{A_1}, s'''_B \rangle) \cdot \mu_{A_1 B-G}(\langle s'''_{A_1}, s'''_B \rangle, \tau^*, \langle s'_{A_1}, s'_B \rangle) \right). \end{aligned} \quad (\text{A.162})$$

In equation A.162, $\mu_{A_1 B-G}(\langle s_{A_1}, s_B \rangle, \tau^*, \langle s''_{A_1}, s''_B \rangle)$ and $\mu_{A_1 B-G}(\langle s'''_{A_1}, s'''_B \rangle, \tau^*, \langle s'_{A_1}, s'_B \rangle)$ denote paths of interleaved transitions labelled with non-synchronisable actions in A_1 and B , while $\mu_{A_1 B-G}(\langle s''_{A_1}, s''_B \rangle, a, \langle s'''_{A_1}, s'''_B \rangle)$ represents a synchronous move by A_1 and B . As a result, we further expand $\mu_{A_1 B-G}(\langle s_{A_1}, s_B \rangle, \tau^* a \tau^*, \langle s'_{A_1}, s'_B \rangle)$ to

$$\begin{aligned} & \sum_{\langle s''_{A_1}, s''_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \\ & \left(\sum_{\langle s'''_{A_1}, s'''_B \rangle \in \mathcal{S}_{A_1 B}} \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'''_{A_1}) \cdot \mu_{B-G}(s''_B, a, s'''_B)}{\eta_{A_1 B}(s''_{A_1}, s''_B)} \cdot \right. \\ & \quad \left. \mu_{A_1-G}(s'''_{A_1}, \tau^*, s'_{A_1}) \cdot \mu_{B-G}(s'''_B, \tau^*, s'_B) \right). \end{aligned} \quad (\text{A.163})$$

By applying the same steps to $\mu_{A_2 B-G}(\langle s_{A_2}, s_B \rangle, \tau^* a \tau^*, \langle s'_{A_2}, s'_B \rangle)$ we derive:

$$\begin{aligned} & \sum_{\langle s''_{A_2}, s''_B \rangle \in \mathcal{S}_{A_2 B}} \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \\ & \left(\sum_{\langle s'''_{A_2}, s'''_B \rangle \in \mathcal{S}_{A_2 B}} \frac{\mu'_{A_2-G}(s''_{A_2}, a, s'''_{A_2}) \cdot \mu_{B-G}(s''_B, a, s'''_B)}{\eta_{A_2 B}(s''_{A_2}, s''_B)} \cdot \right. \\ & \quad \left. \mu_{A_2-G}(s'''_{A_2}, \tau^*, s'_{A_2}) \cdot \mu_{B-G}(s'''_B, \tau^*, s'_B) \right). \end{aligned} \quad (\text{A.164})$$

We now consider the case when the sequence $\tau^* a \tau^*$ does not contain deleted transitions. Given the rules A.158 for constructing $\mathcal{R}'$, we obtain that $\langle s''_{A_1}, s''_B \rangle = \langle s'_{A_1}, s'_B \rangle$ and the suffix sequence τ^* is an empty sequence ϵ . Therefore, we simplify the previous equations A.163 and A.164 into:

$$\sum_{\langle s''_{A_1}, s_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1}) \cdot \mu_{B-G}(s''_B, a, s'_B)}{\eta_{A_1 B}(s''_{A_1}, s''_B)}, \text{ and} \quad (\text{A.165})$$

$$\sum_{\langle s''_{A_2}, s_B \rangle \in \mathcal{S}_{A_2 B}} \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \frac{\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) \cdot \mu_{B-G}(s''_B, a, s'_B)}{\eta_{A_2 B}(s''_{A_2}, s''_B)}. \quad (\text{A.166})$$

Given that $A_1 \approx A_2$, then $\mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) = \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2})$, then we verify the condition A.160 as

$$\mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) = \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B).$$

Consequently, to verify the condition for weak bisimulation define by equation A.159 we need to show that

$$\frac{\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1 B}(s''_{A_1}, s''_B)} = \frac{\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2})}{\eta_{A_2 B}(s''_{A_2}, s''_B)}. \quad (\text{A.167})$$

We recall the normalisation rules associated with the parallel composition operator defined in Chapter 3, where the adjusted transition is defined as follows:

$$\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1}) = \begin{cases} \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1}) & \text{if } a \in \mathcal{E}_{A_1}^{int} \\ \frac{\mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1 B}^{enabled}} & \text{if } a \in \mathcal{E}_{A_1}^{out} \end{cases}$$

where

$$\eta_{A_1 B}^{enabled}(s''_{A_1}, s_B) = \frac{\mu_{A_1}^{enabled}(s''_{A_1}, s_B)}{\sum_{(s''_{A_1}, a^{out}, s'_{A_1})} \mu_{A_1}(s''_{A_1}, a^{out}, s'_{A_1})}$$

Given that there are no deleted transition in the sequence $\tau^* a$ in A_1 , we have that $(s''_{A_1}, s''_{A_2}) \in \mathcal{R}$, $(s'_{A_1}, s'_{A_2}) \in \mathcal{R}$ and the transitions of s''_{A_1} are all preserved in A_2 from state s''_{A_2} . Therefore, the equation A.167 is verified in this case.

In the case that the prefix sequence τ^* in A_1 contains deleted transitions, if the deleted transitions are not incoming transitions to (s''_{A_1}, a, s'_{A_1}) the previous steps can be applied to verify the condition for weak bisimulation define by equation A.159 as $A_1 \approx A_2$, implies that $\mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) = \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2})$.

On the other hand, two additional cases need to be considered if the delete transitions are the last of prefix sequence τ^* in A_1 . In the first case we consider that all output actions from state s''_{A_1} can be synchronised. As a result, $\eta_{A_1 B}^{enabled}(s''_{A_1}, s_B) = 1$ and $\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1}) = \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})$. In this case,

$$\eta_{A_1 B}(s''_{A_1}, s''_B) = \sum_{\substack{(s''_{A_1}, a, s'_{A_1}) \in \Delta_A(s''_{A_1}) \\ a \in \mathcal{E}_{A_1}^{loc}}} \mu_{A_1}(s''_{A_1}, a, s'_{A_1}) + \sum_{\substack{(s''_B, b, s'_B) \in \Delta_B(s''_B) \\ b \in \mathcal{E}_B^{loc}}} \mu_B(s''_B, b, s'_B)$$

and

$$\eta_{A_2 B}(s''_{A_2}, s''_B) = \sum_{\substack{(s''_{A_2}, a, s'_{A_2}) \in \Delta_A(s''_{A_2}) \\ a \in \mathcal{E}_{A_2}^{loc}}} \mu_{A_2}(s''_{A_2}, a, s'_{A_2}) + \sum_{\substack{(s''_B, b, s'_B) \in \Delta_B(s''_B) \\ b \in \mathcal{E}_B^{loc}}} \mu_B(s''_B, b, s'_B).$$

Given that there are no blocked transitions from both states s''_{A_1} and s''_{A_2} , then we have

$$\sum_{\substack{(s''_{A_1}, a, s'_{A_1}) \in \Delta_A(s''_{A_1}) \\ a \in \mathcal{E}_{A_1}^{loc}}} \mu_{A_1}(s''_{A_1}, a, s'_{A_1}) = \sum_{\substack{(s''_{A_2}, a, s'_{A_2}) \in \Delta_A(s''_{A_2}) \\ a \in \mathcal{E}_{A_2}^{loc}}} \mu_{A_2}(s''_{A_2}, a, s'_{A_2}).$$

Therefore, in order to verify the equations A.165 and A.166 result in the same value, we have to show that

$$\begin{aligned} \sum_{\langle s''_{A_1}, s''_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1}) &= \\ \sum_{\langle s''_{A_2}, s''_B \rangle \in \mathcal{S}_{A_2 B}} \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \mu_{A_2-G}(s''_{A_2}, a, s'_{A_2}) & \end{aligned} \quad (\text{A.168})$$

We start by expanding the left side as

$$\sum_{\langle s''_{A_1}, s''_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s'''_{A_1}) \cdot \left(\sum_{\langle s''_{A_1}, s''_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1}) \right). \quad (\text{A.169})$$

In order for $\sum_{\langle s''_{A_1}, s_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1})$ to be propagated forward to $\mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})$, we are either in the case of transformation T_1 and transformation T_2 when all the incoming transitions to state s''_{A_1} come from state s'''_{A_1} . In both cases we have that $(s'''_{A_1}, s''_{A_2}) \in \mathcal{R}$ and

$$\mu_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1}),$$

which verifies the equality A.168.

We now consider the case where some transitions labelled with output actions in A_1 and A_2 may be blocked as B is not ready to synchronise with all of them, therefore $\eta_{A_1 B}^{enabled}(s''_{A_1}, s_B) < 1$, $\eta_{A_2 B}^{enabled}(s''_{A_2}, s_B) < 1$ and $\eta_{A_1 B}^{enabled}(s''_{A_1}, s_B) \neq \eta_{A_2 B}^{enabled}(s''_{A_2}, s_B)$. We start by simplifying the equations A.165 and A.166 by removing the probabilities of transitions in B as these are not changed:

$$\sum_{\langle s''_{A_1}, s''_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1 B}(s''_{A_1}, s''_{B_1})}, \text{ and} \quad (\text{A.170})$$

$$\sum_{\langle s''_{A_2}, s''_{B_1} \rangle \in \mathcal{S}_{A_2 B}} \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \frac{\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2})}{\eta_{A_2 B}(s''_{A_2}, s''_{B_1})}. \quad (\text{A.171})$$

By applying the expansion rule to the sequence τ^* in equation A.170 we obtain

$$\sum_{\langle s'''_{A_1}, s''_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s'''_{A_1}) \cdot \left(\sum_{\langle s''_{A_1}, s''_{B_1} \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1 B}(s''_{A_1}, s''_{B_1})} \right), \quad (\text{A.172})$$

where

$$\mu'_{A_1-G}(s''_{A_1}, a, s'_{A_1}) = \frac{\mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1 B}^{enabled}(s''_{A_1}, s_B)}$$

Given that the transitions $s'''_{A_1}, \tau, s''_{A_1}$ have been propagated forward when constructing A_2 from A_1 , we derive the following from equation A.171:

$$\mu_{A_1-G}(s_{A_1}, \tau^*, s'''_{A_1}) = \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}), \quad (\text{A.173})$$

as $A_1 \approx A_2$. Therefore, we need to show the following condition holds in order to show that the expressions in equations A.170 and A.171 result in the same value.

$$\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \sum_{\langle s''_{A_1}, s''_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu'_{A_1-G}(s'''_{A_1}, a, s'_{A_1}). \quad (\text{A.174})$$

We start by expanding $\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2})$ into

$$\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \frac{\mu_{A_2-G}(s''_{A_2}, a, s'_{A_2})}{\eta_{A_2 B}^{enabled}(s''_{A_2}, s_B)} \quad (\text{A.175})$$

In order for $\sum_{\langle s''_{A_1}, s_B \rangle \in \mathcal{S}_{A_1 B}} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1})$ to be propagated forward to $\mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})$, we are either in the case of transformation T_1 and transformation T_2 when all the incoming transitions to state s''_{A_1} come from state s'''_{A_i} . In both cases we have that $(s'''_{A_1}, s''_{A_2}) \in \mathcal{R}$ and

$$\mu_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1}). \quad (\text{A.176})$$

We then use this equality to replace derive the following from equation A.175:

$$\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \frac{\mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_2 B}^{enabled}(s''_{A_2}, s_B)}, \quad (\text{A.177})$$

where

$$\eta_{A_2 B}^{enabled} = \frac{\mu_{A_2}^{enabled}(s''_{A_2}, s_B)}{\sum_{(s''_{A_2}, a^{out}, s'_{A_2})} \mu_{A_2}(s''_{A_2}, a^{out}, s'_{A_2})}.$$

Based on the equality A.183 we further expand $\eta_{A_2 B}^{enabled}$ as follows

$$\eta_{A_2 B}^{enabled} = \frac{\mu_{A_2}^{enabled}(s''_{A_2}, s_B)}{\sum_{(s''_{A_1}, a^{out}, s'_{A_1})} \mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a^{out}, s'_{A_1})}. \quad (\text{A.178})$$

From the condition for propagation rules defined by the reduction algorithm, the set of output actions that are enabled at state s''_{A_2} are the same as the actions enabled at state s''_{A_1} . Consequently we redefine $\eta_{A_2B}^{enabled}$ as follows

$$\eta_{A_2B}^{enabled}(s''_{A_2}, s''_B) = \frac{\mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1}^{enabled}(s''_{A_1}, s_B)}{\mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \left(\sum_{(s''_{A_1}, a^{out}, s'_{A_1})} \mu_{A_1-G}(s''_{A_1}, a^{out}, s'_{A_1}) \right)}, \quad (\text{A.179})$$

which is the same as $\eta_{A_1B}^{enabled}(s''_{A_1}, s''_B)$.

Therefore, we can conclude that

$$\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \frac{\mu_{A_1-G}(s'''_{A_1}, \tau, s''_{A_1}) \cdot \mu_{A_1-G}(s''_{A_1}, a, s'_{A_1})}{\eta_{A_1B}^{enabled}(s''_{A_1}, s''_B)}, \quad (\text{A.180})$$

We now consider the last case where the deleted transitions are the outgoing transitions from state s'''_{A_1} in equation A.163 defined earlier.

$$\begin{aligned} & \sum_{\langle s''_{A_1}, s_B \rangle \in \mathcal{S}_{A_1B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \\ & \left(\sum_{\langle s'''_{A_1}, s'''_B \rangle \in \mathcal{S}_{A_1B}} \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'''_{A_1}) \cdot \mu_{B-G}(s''_B, a, s'''_B)}{\eta_{A_1B}(s''_{A_1}, s''_B)} \cdot \right. \\ & \left. \mu_{A_1-G}(s'''_{A_1}, \tau^*, s'_{A_1}) \cdot \mu_{B-G}(s'''_B, \tau^*, s'_B) \right). \end{aligned}$$

From the propagation rule A.100 defined for transformation T_2 and the rules for constructing $\mathcal{R}'$, we derive that the suffix sequence τ^* for B is an empty sequence ϵ , as $s'''_B = s'_B$, and a sequence of a single non-synchronisable action τ for A_1 . We simplify the previous equation as follows:

$$\begin{aligned} & \sum_{\langle s''_{A_1}, s_B \rangle \in \mathcal{S}_{A_1B}} \mu_{A_1-G}(s_{A_1}, \tau^*, s''_{A_1}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \quad (\text{A.181}) \\ & \left(\sum_{\langle s'''_{A_1}, s'''_B \rangle \in \mathcal{S}_{A_1B}} \frac{\mu'_{A_1-G}(s''_{A_1}, a, s'''_{A_1}) \cdot \mu_{B-G}(s''_B, a, s'''_B)}{\eta_{A_1B}(s''_{A_1}, s''_B)} \cdot \mu_{A_1-G}(s'''_{A_1}, \tau, s'_{A_1}) \right) \cdot \end{aligned}$$

By applying the same principles to equation A.164 we obtain the following

$$\sum_{\langle s''_{A_2}, s_B \rangle \in \mathcal{S}_{A_2 B}} \mu_{A_2-G}(s_{A_2}, \tau^*, s''_{A_2}) \cdot \mu_{B-G}(s_B, \tau^*, s''_B) \cdot \frac{\mu'_{A_2-G}(s''_{A_2}, a, s'_{A_2}) \cdot \mu_{B-G}(s''_B, a, s'_B)}{\eta_{A_2 B}(s''_{A_2}, s''_B)}. \quad (\text{A.182})$$

In order for $\mu_{A_1-G}(s'''_{A_1}, \tau, s'_{A_1})$ to be propagated backwards to

$$\sum_{\langle s'''_{A_1}, s'''_B \rangle \in \mathcal{S}_{A_1 B}} \mu'_{A_1-G}(s'''_{A_1}, a, s'''_{A_1})$$

using transformation T_2 , all the incoming transitions to state s''_{A_1} originate from state s'''_{A_i} , which reduces the summation to only one state s'''_{A_i} .

From the propagation rule A.109 defined for transformation T_2 and the rules for constructing $\mathcal{R}'$, we obtain that $(s'_{A_2}, s'_{A_1}) \in \mathcal{R}'$ and

$$\mu_{A_2-G}(s''_{A_2}, a, s'_{A_2}) = \mu_{A_1-G}(s''_{A_1}, a, s'''_{A_1}) \cdot \mu_{A_1-G}(s'''_{A_1}, \tau, s'_{A_1}). \quad (\text{A.183})$$

We have therefore shown that $\mathcal{R}'$ is a weak bisimulation on $\hat{A}_{\parallel}$, such that $\forall (s, t) \in \mathcal{R}', \forall C \in \hat{\mathcal{S}}_C$ and $\forall a \in \text{shared}(A_1, B) = \mathcal{E}_{A_1} \cap \mathcal{E}_B$ the following condition holds the condition A.159:

$$\begin{aligned} \hat{\mu}_G(s, \tau^* a \tau^*, C) &= \hat{\mu}_G(t, \tau^* a \tau^*, C), \\ \hat{\mu}_G(s, \tau^*, C) &= \hat{\mu}_G(t, \tau^*, C) \text{ where } \tau \in (\mathcal{E}_{A_1} \cup \mathcal{E}_B) \setminus \text{shared}(A_1, B). \end{aligned}$$

A.6. Summary

In this Chapter we have defined notions of equivalence between PCA models by extending the definitions for strong and weak bisimulations that have been applied to PLTS models. We then used the definition of weak bisimulation for PCA models to show that the models produced by the reduction algorithm described in the Chapter 4 preserve the reachability properties of the original representations. In order to allow compositional system verification, we show that weak bisimulation is a congruence for the parallel composition operator. This means that composite representations constructed using the original models or their reduced representations are equivalence w.r.t. their reachability probabilistic properties. We have also shown that

weak bisimulation is a congruence for all the other operators supported by PCA. As a result, given an expression involving PCA models and any combination of the supported operators, the models involved can be substituted by reduced representations whilst preserving the reachability probabilistic properties of the original result.

B. Tool Support - LTSA-PCA

We present in this Chapter our extension of the LTSA model checker [MK06]. It supports the specification, visualisation and failure analysis of composable, probabilistic behaviour of component-based systems, modelled as Probabilistic Component Automata (PCA). To evaluate aspects such as the probability of system failure, a DTMC model can be automatically constructed from the composition of the PCA representations of each component and analysed in tools such as PRISM [KNP]. Existing behavioural analysis techniques in LTSA can be applied to PCA representations to verify, for instance, the compatibility of interface behaviour between components with matching provided-required interfaces.

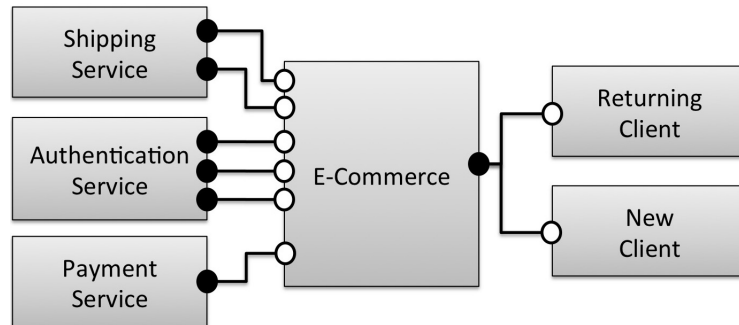


Figure B.1.: Architecture of e-commerce System [FGT12]

We use the e-commerce system described by Filieri *et. al* [FGT12] to illustrate the LTSA-PCA tool. This system consists of a web-service that sells merchandise and integrates three external web-services: authentication, shipping and payment. The original DTMC model [FGT12] is a closed representation of the entire system and has not been constructed from the models of its components. To illustrate our approach, we assume that the system is constructed from the components shown in Figure B.1. Based on the original DTMC model (Figure B.2) we specify the behaviour of each component in P-FSP. We then construct the composite model of the e-commerce system and show it is equivalent to the original DTMC model, thereby producing the same reliability analysis.

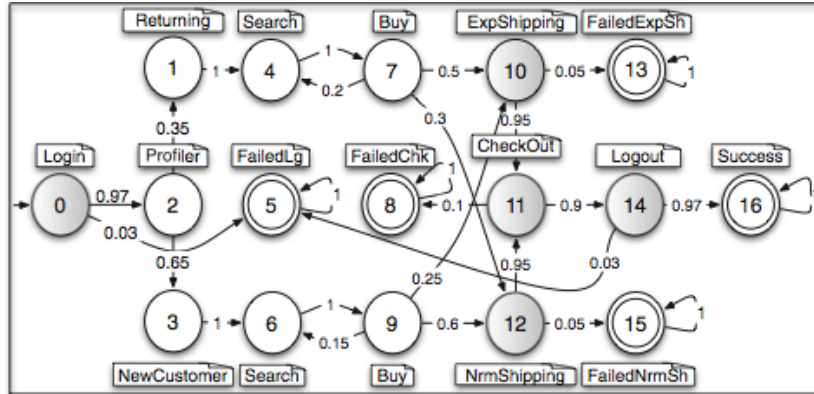


Figure B.2.: Original DTMC Model of e-commerce System [FGT12]

In the same way that LTS models are defined based on FSP expressions, PCA models are constructed using our probabilistic extension of FSP (P-FSP). The specification of LTS or PCA models is done using the *Editor* panel shown in Figure B.3.

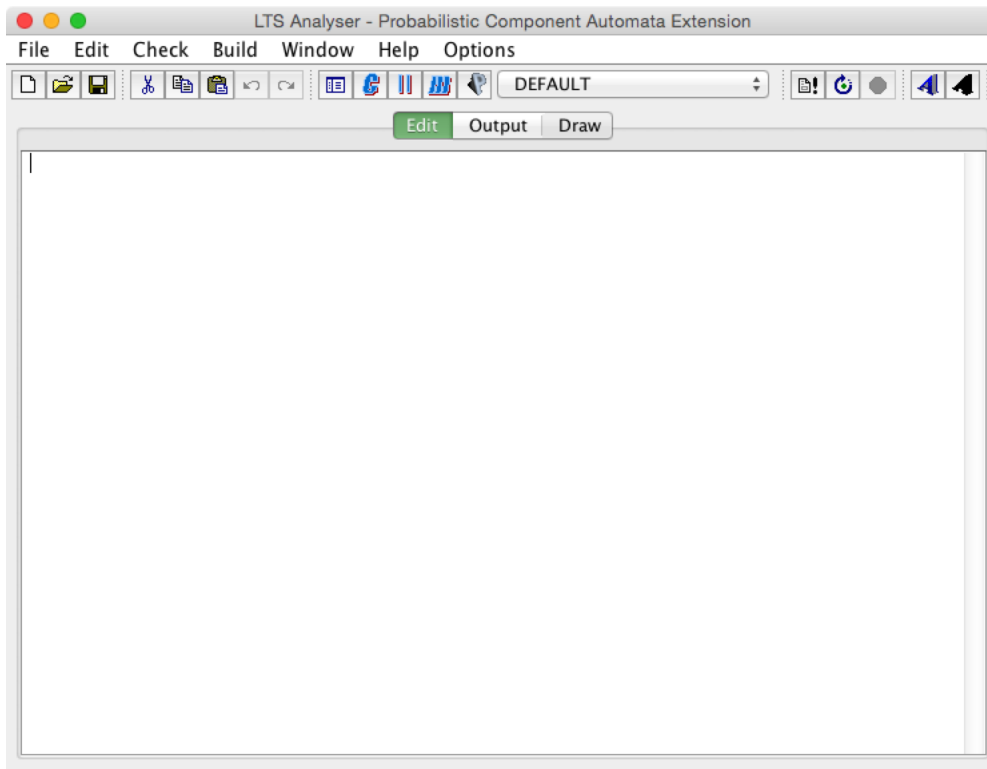


Figure B.3.: Editor Panel - LTSA-PCA

The P-FSP expression for the Authentication Service is shown in Figure B.4. Note that the user has to specify the key-word `pca` before specifying any P-FSP expression. This allows us to support both the specification of PCA models and LTS models, though only one type of model can be specified when modelling a particular system.

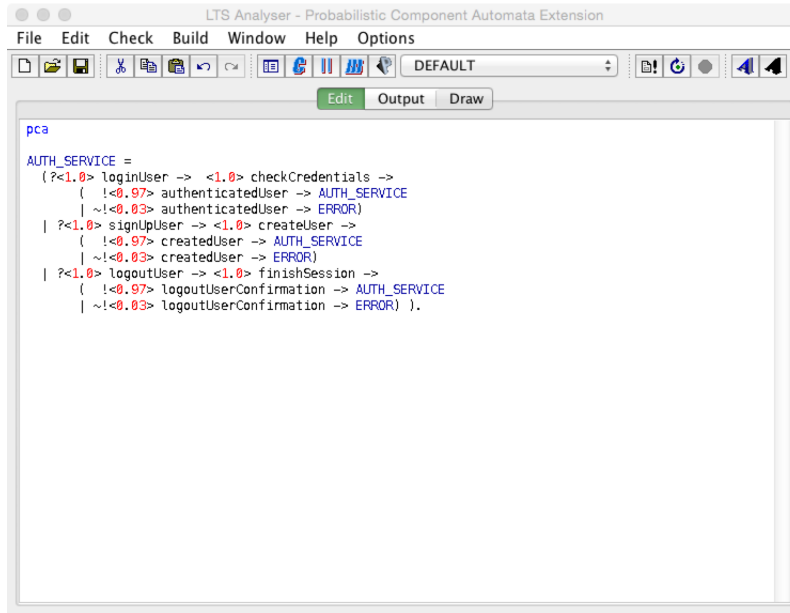


Figure B.4.: Editor Panel - Authentication Service P-FSP

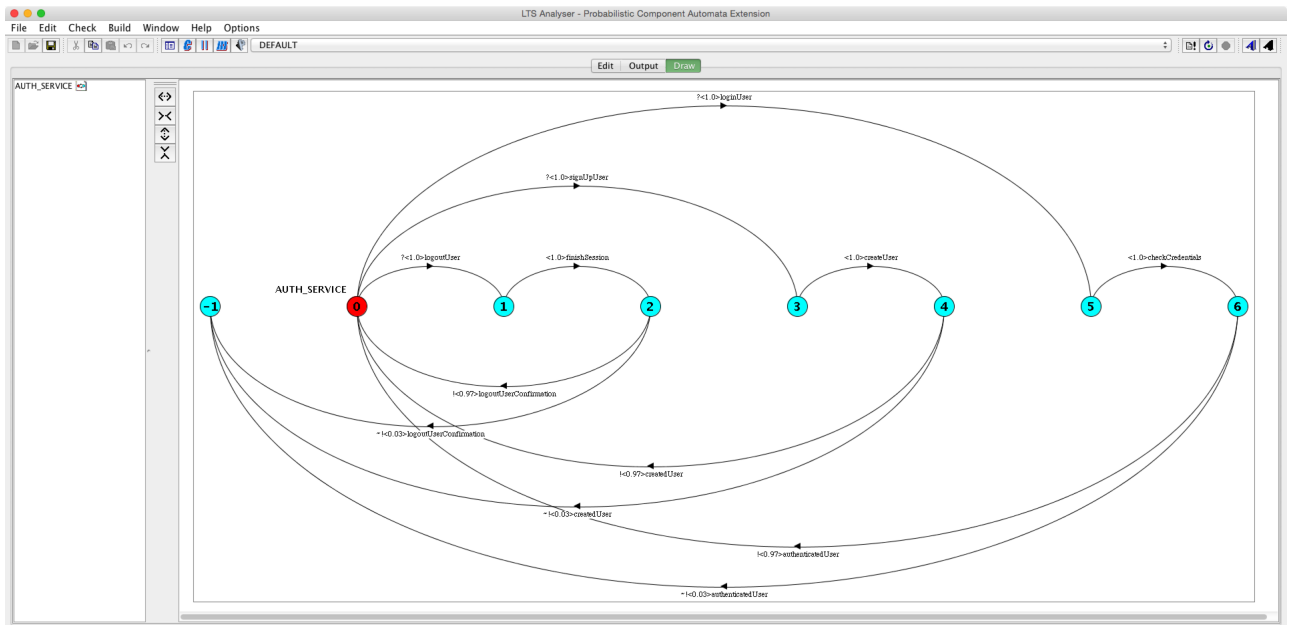


Figure B.5.: Editor Panel - Authentication Service PCA

The corresponding PCA model for the Authentication Service is constructed by clicking on the *C* button on top of the **Edit** panel to compile the P-FSP expression. The PCA graphical representation of the Authentication Service can be seen using the **Draw** panel, as depicted by Figure B.5.

The behaviour of the Authentication Service denotes three provided interfaces with no shared (internal) behaviour, which is specified using a single simple process. In order to avoid duplicate specification of shared behaviour, the system designer can define a local process which can be re-used in different parts of the main model. These local processes can be used to model private functions of a class as their scope is local to the main process. While a simple single process ends with a dot, a comma separates the behaviour of the main process specification from its local processes.

Consider now the behaviour of the Shipping Service where its two provided interfaces send back the same message after processing each request (**shippingConfirmation**). The local process **CONFIRMATION** represents the shared behaviour, thereby avoiding duplicating the specification of the behaviour after actions **processNormalRequest** and **processExpressRequest**.

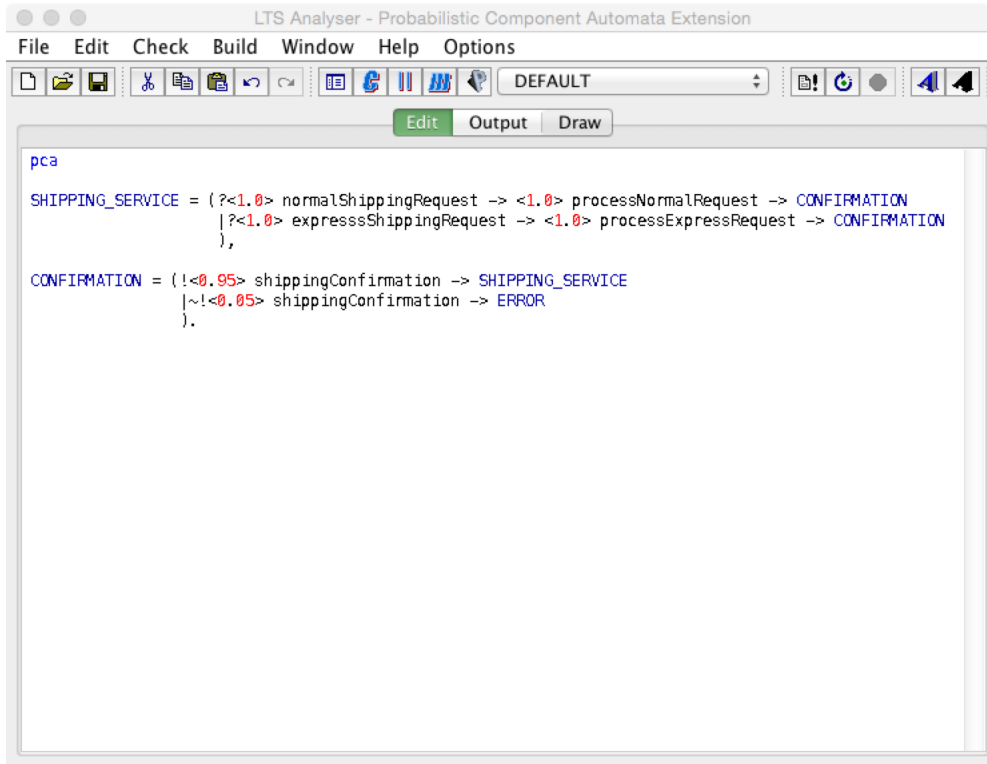


Figure B.6.: Editor Panel - Shipping Service P-FSP

We now focus on how the composite model for the e-commerce system is built. The behaviour of the E-Commerce component has to be adjusted to support interaction with the two clients, as shown in Listing B.3.

Listing B.1: E-Commerce component adjusted for two clients

```
|| E_COMMERCE.TWO_CLIENTS = (E_COMMERCE / {
    {new.signUp}/signUp,
    {ret.login}/login,
    {new.confirmation, ret.confirmation}/confirmation,
    {new.searchProduct, ret.searchProduct}/searchProduct,
    {new.buy, ret.buy}/buy,
    {new.continueSearch, ret.continueSearch}/continueSearch,
    {new.normalShipping, ret.normalShipping}/normalShipping,
    {new.expressShipping, ret.expressShipping}/expressShipping,
    {new.productsShipped, ret.productsShipped}/productsShipped,
    {new.checkout, ret.checkout}/checkout,
    {new.logout, ret.logout}/logout,
    {new.logoutConfirmation, ret.logoutConfirmation}/logoutConfirmation } ).
```

Similarly, all the actions of the two clients need to be prefixed to ensure that the behaviour of the two clients is fully distinguished (Listing B.2). Note that the processes `E_COMMERCE.TWO_CLIENTS`, `RET_CLIENT` and `N_CLIENT` are defined based on processes that had been previously defined, *e.g.* `E_COMMERCE`. Consequently, their name needs to be preceded by the symbol `||`.

Listing B.2: Relabelling of New-Client and Returning-Client components

```
|| RET_CLIENT = ret:RETURNING_CLIENT.
|| N_CLIENT = new:NEW_CLIENT.
```

The final composite model of the e-Commerce system (`E_COMMERCE_SYS`) is constructed by composing the adjusted models `E_COMMERCE.TWO_CLIENTS`, `RET_CLIENT` and `N_CLIENT` with the models for the Authentication, Shipping and Payment services. This is done by clicking on the parallel composition button `||` below the main menu bar. Note that only one composite model is constructed when clicking on the parallel composition button. If the current file being edited had multiple composite models, the dropdown bar below above the `Edit`, `Output` and `Draw` buttons can be used to select the active composite model.

When defining the expression for constructing the composite model of the e-Commerce system, a special version of the parallel composition operator is used to determine the proportion of requests from each client: `((RET_CLIENT, 0.65);(N_CLIENT, 0.35) || E_COMMERCE.TWO_CLIENTS)`. The last step for building the composite model involves using the hiding operator `\` to remove

the internal actions related to interactions with the different service components which were not present in the original DTMC model.

Listing B.3: E.Commerce component adjusted for two clients

```
|| E.COMMERCE.SYS =
  ( ([ (RET.CLIENT, 0.65); (N.CLIENT, 0.35) || E.COMMERCE.TWO.CLIENTS])
    || AUTH.SERVICE || SHIPPING.SERVICE || PAYMENT.SERVICE)
  \ { authenticatedUser, checkCredentials, confirmPayment, createUser,
    createdUser, expressShippingRequest, finishSession, loginUser,
    logoutUser, logoutUserConfirmation, normalShippingRequest,
    processExpressRequest, processNormalRequest, registerPayment,
    registerProduct, shippingConfirmation, signUpUser } ).
```

The composite model before removing those transitions has 47 states and 60 transitions. Although it is still possible to view its graphical representation in the **Draw** panel, a textual representation can be analysed in the **Transitions** panel as show in Figure B.7, which is activated via the Window option in the main menu bar.

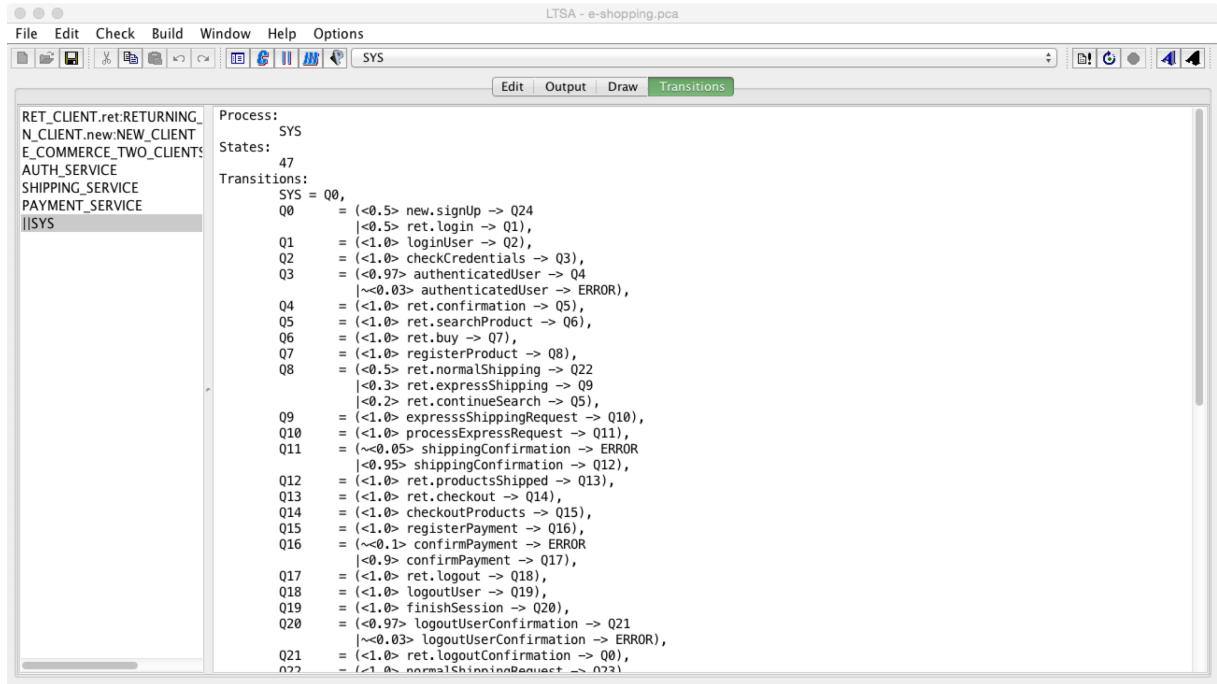


Figure B.7.: Transitions Panel - LTSA-PCA

Furthermore, the **Alphabet** panel shows the alphabet of each model used in the currently active composite model. This panel can be used to analyse if all actions have been synchronised in the composite model.

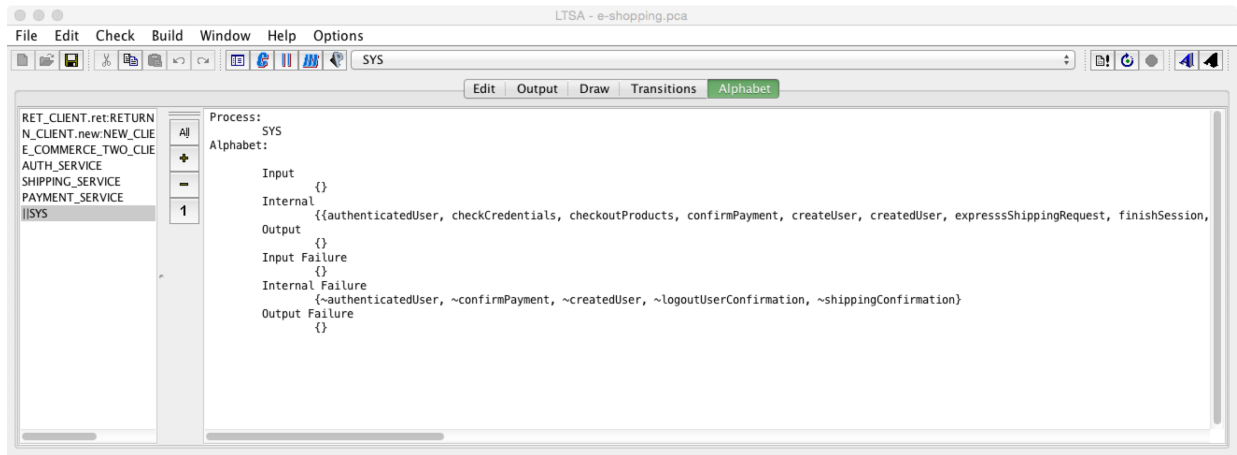


Figure B.8.: Alphabet Panel - LTSA-PCA

After the composite model of the e-commerce system has been constructed, it can be converted to a DTMC model for analysis of reliability properties in PRISM model checker as it does not have input actions. This is done by clicking on the **Prism** button next to the parallel composition $\parallel$ button. The converted DTMC model is shown in the **PRISM** panel, as depicted in Figure B.9, and can then be copied to PRISM model checker for analysis. The tool currently supports automatic generation of state variables for reliability analysis: one for each failure action and a general **failure** variable that denotes the execution of any failure. The user is required to manually edit the model in PRISM for analysis of operational properties that do not involve failure actions.

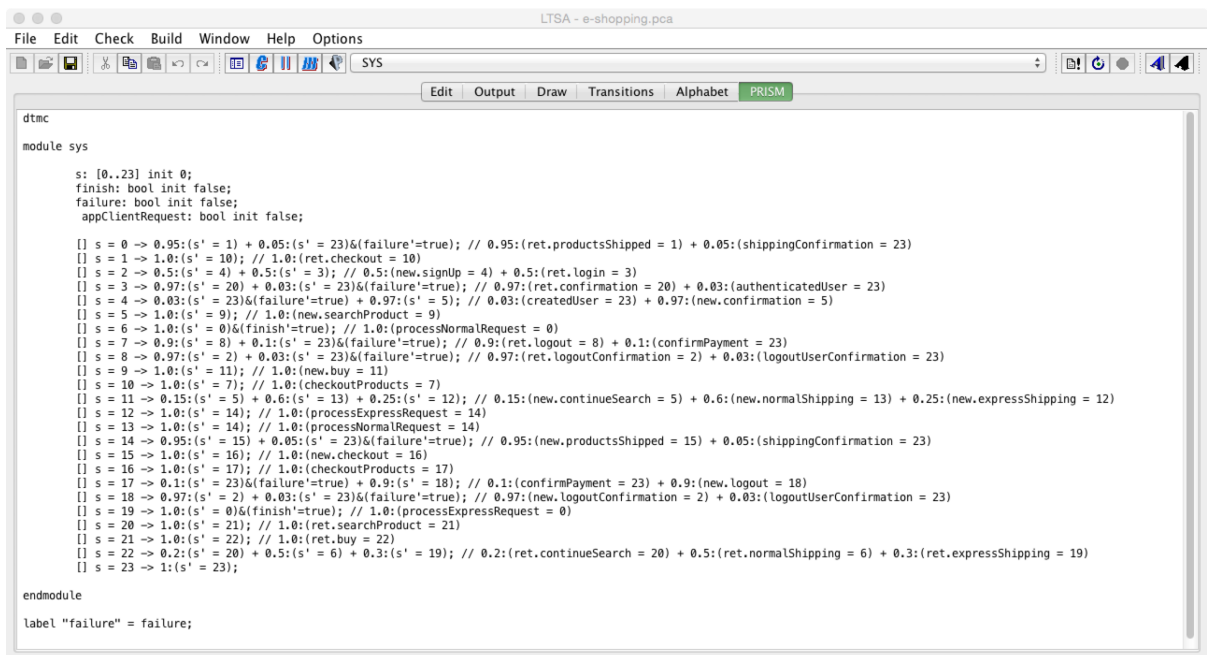


Figure B.9.: PRISM Panel - LTSA-PCA