Compositional Reliability Analysis using Probabilistic Component Automata

Pedro Rodrigues, Emil Lupu, Jeff Kramer
Department of Computing,
Imperial College London,
London SW7 2AZ, UK

Abstract

Compositionality is a key property in the development and analysis of component-based systems. In non-probabilistic formalisms such as Labelled Transition Systems (LTS) the functional behaviour of a system can be readily constructed from the behaviours of its parts. However, this is not true for probabilistic extensions of LTS, which are necessary to analyse non-functional properties such as reliability. We propose Probabilistic Component Automata (PCA) as a probabilistic extension to Interface Automata to automatically construct a system model by composing models of its sub-components. In particular, we focus on modelling failure scenarios, failure handling and failure propagation. Additionally, we propose a novel algorithm based on Compositional Reachability Analysis to mitigate the well-known state-explosion problem associated with composable models. Both Probabilistic Component Automata and the reduction algorithm have been implemented in the LTSA tool.

1 Introduction

Many software systems, from service-based to ubiquitous systems, are built by combining new services and components with existing ones. Models for these systems should therefore preserve the modularity and reusability properties of component-based design and accommodate compositional analysis. This has been successfully explored for functional behaviour models using LTS representations [1–4] and software architectures [5–7]. However, this is less well supported for non-functional properties such as reliability and performance which require representations that consider time and probabilistic

information. Although performance models are compositional [8], their semantics is based on the duration of actions. Consequently, these models allow to answer 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 on probabilistic compositional reachability analysis of failure states to answer questions such as "what is the probability that the system fails?" or "what is the probability of failure after action a?".

Discrete-Time Markov Chains (DTMCs) have been traditionally used for probabilistic reachability analysis [9–12]. However, a composite model of a system cannot be simply constructed from the DTMC models of its subcomponents, *i.e.* the models are not composable. This limitation arises from the difficulty of composing probabilistic behaviour. For example, Probabilistic LTS (PLTS) have been advocated for reasoning about non-functional properties [13]. However, the composition of two PLTS may result in a model whose probabilistic choices from a given state do not sum to 1 [14] and a straightforward normalisation does not always accurately capture the composite behaviour. Probabilistic I/O Automata (PIOA) [15] and Probabilistic Interface Component Protocols (PCIP) [16] attempt to address this but their semantics introduces additional characteristics (e.g., input-enabledness and error-on-wait) which hinder their applicability. Furthermore, these models do not support the representation of failure scenarios and failure handling.

We propose a modelling formalism, Probabilistic Component Automata (PCA), that is compositional and includes primitives to represent failure scenarios, failure propagation and failure handling. Our model complements Architectural Description Languages, such as Darwin [6], to describe the system behaviour and how exceptions are used to deal with failures in object-oriented languages. As composite PCA models may suffer from state-explosion, we extend the use of Compositional Reachability Analysis [17] to hide probabilistic transitions in PCA and perform compositional reliability analysis. The main contributions of the paper, PCA and the associated CRA algorithm, are described in Sections 3 and 4. In this report, we also: (a) conduct a brief review (Section 2) of existing formalisms for modelling probabilistic behaviour, their inter-relationships and limitations, (b) evaluate the benefits of PCA in conjunction with the reduction algorithm for scalable reliability analysis (Section 5). Conclusions and future work are presented in Section 6.

2 Related Work

2.1 Non-composable models

Discrete-Time Markov Chains (DTMCs) have been initially proposed by Cheung [9] to represent the reliability of component-based systems. The behaviour of component C_i is modelled as a single state s_i, denoting component C_i is executing. A transition matrix P represents the transfer of execution control between components, where $P(s_i, s_j)$ denotes the probability of transferring control from C_i to C_j, conditional on the successful execution of C_i. If C_i fails, a transition to a global failure state F is added to state s_i and $P(s_i, F)$ denotes the probability of such failure. A final absorbing state C represents the successful termination.

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. Therefore, when the system architecture changes a new representation has to be manually defined and the system needs to be profiled again to obtain a new transition matrix P. 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. Filieri *et. al* [10] extended Cheung's approach to allow failures to be *propagated* but their approach still does not support automatic construction of the model from the representations of its parts. To overcome this, Wang *et. al* [11] defined mappings between architectural patterns and DTMC representations of the entire system. However, such mappings must be manually defined and the resulting models do not include the internal behaviour of each component.

2.2 Composable Models

Probabilistic I/O Automata (PIOA) [15] are a probabilistic extension of I/O Automata [18] that distinguish between *input actions*, which follow reactive semantics and *internal/output actions*, which follow generative semantics. When composing two PIOA, only matching pairs of input-output actions are synchronised, which correspond to bindings between the *provided* and *required* interfaces of components. *Internal actions* represent behaviour that is not externally visible to other components. Distinguishing between input and output actions allows PIOA to address some of the inconsistencies encountered in probabilistic LTS [13]. However, PIOA are *input-enabled* models, *i.e.* each component must process any input action at any time, re-

gardless of its internal state. This makes PIOA unsuitable for representing component-based software systems.

Interface Automata (IA) [19] are similar to I/O Automata but do not require input-enabledness. In essence, an automaton that executes an output action waits for the automaton with the corresponding input action to be ready to communicate. This is achieved by removing *illegal* transitions from the product of the two automata, *i.e.* where this property is not verified. However, IA models do not consider probabilistic information.

Probabilistic Component Interface Protocols (PCIP) [16] are a full probabilistic extension[1] to IA, but a different semantics is used to construct composite models. While in IA the transitions in the composite automaton associated with illegal behaviour are discarded to implement a *wait on call* semantics, in PCIP such situations are considered erroneous behaviour and are included in the composite PCIP as transitions leading to a special *error state*. This hinders their applicability for reliability analysis where transitions to the error state represent failures of actions, *e.g.* communication failures.

Other approaches to reliability analysis based on the probability of reaching an *error state* as a result of failures are described in existing surveys [21, 22]. However, a compositional model that allows for representation for failures, failure propagation and failure handling is still missing.

3 Probabilistic Component Automata

We define Probabilistic Component Automata (PCA) as a probabilistic extension to IA [19] with support for the representation of failures. Probabilistic information is added to the transitions between states and we redefine accordingly the semantics of the operators to construct single and composite models. In composite models we follow the *wait on call* semantics of IA and introduce a different, and arguably more intuitive, normalisation than the one used in PIOA and PCIP. We further introduce an explicit representation for failure actions and failure handling actions that is analogous to the conventional use of exceptions in object-oriented programming languages. Our model has been implemented as an extension to the LTSA tool [1] and is available at *https://wp.doc.ic.ac.uk/dse/software/ltsa-pca/*. The implementation aspects are described in [23].

[1]Probabilistic Interface Automata [20] only support the composition of a probabilistic model of the environment with a non-probabilistic model of the software system.

3.1 Definition

A Probabilistic Component Automaton is defined as $P = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ where:

- $\mathcal{S}$ is a 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 $\mathcal{E}^{int}$ and $\mathcal{E}^{out}$ are internal actions and output actions, respectively;
- $\Delta \subseteq (\mathcal{S} \times \mathcal{E} \times \mathcal{S})$ is the set of transitions.
- $\mu : \Delta \to [0, 1]$ where $\mu = p(s', a \mid s')$ denotes the probability of reaching state s' from state s through the execution of action a.

Similarly to IA, *input* actions model the receiving end of a communication channel or (interface) methods that can be called, whereas *output* actions model the invocation of methods or sending of messages. Hereafter we refer to both types of interactions as communication between components.

3.2 Modelling Basic Components

Just as Finite State Processes are used to specify Labelled Transitions Systems, Probabilistic Finite State Processes (P-FSP) [23] support incremental specification of PCA models. The correspondence between P-FSP expressions and PCA models is given by the function $pca : E \to \text{PCA}$. 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 2.

$$\frac{}{(a, p_a \to E) \xrightarrow{a, p_a} E} \qquad \text{(Rule 1)}$$

A transition consists of *a)* an action type: ? for input, ! for output, no-symbol for internal, $\sim$ for internal failures, $\sim$? for input failures and $\sim$! for output failures; *b)* the execution probability p, and *c)* the action label $a \in \mathcal{E}$. The corresponding PCA is given by $pca(a, p_a \to E) = \langle \mathcal{S} \cup \{p\},\ p,\ \mathcal{E} \cup \{(p, a, q)\},\ \mu \cup \{(p, a, q) \to p_a\} \rangle$.

$$\frac{}{(p_1\ a_1 \mathrel{->} E_1 \mid \ldots \mid p_n\ a_n \mathrel{->} E_n) \xrightarrow{a_i, p_{a_i}} E_i} \qquad \text{(Rule 2)}$$

The *choice* operator defines possible outcomes from a given state. If $a_1, \ldots, a_n$ are locally controlled actions (internal or output), then $(p_1\ a_1\ {->}\ E_1 \,|\, \ldots \,|\, p_n\ a_n\ {->}\ E_n)$ describes a PCA that initially engages in any action a_i with probability p_i. In this case, the choice operator can be used to model `if` or `switch` statements. On the other hand, if $a_1, \ldots, a_n$ are input actions, the action a_i the PCA engages in is dictated by the environment, *i.e.* other processes that output a_i. As input actions follow reactive semantics, the probabilities associated with transitions with the same action label are equal to 1. Once an input label is chosen, the process can choose locally different outcomes with different probabilities. This case models how the provided interfaces of a component are used by other components, which is only known in a specific architectural configuration i.e., when the components are bound. The corresponding PCA model is formally defined by the following. Let $1 \leq i \leq n$ and $pca(E_i) = \langle \mathcal{S}_i,\, q_i, \mathcal{E}_i, \Delta_i, \mu_i \rangle$, then $pca(\rho_1\ a_1\ {->}\ E_1 \,|\, \ldots \,|\ \rho_n\ a_n\ {->}\ E_n) = \langle (\bigcup_i \mathcal{S}_i) \cup \{p\},\, p, (\bigcup_i \mathcal{E}_i) \cup \{a_1, \ldots, a_n\}, (\bigcup_i \Delta_i) \cup \{(p, a_i, q_i)\},\ (\bigcup_i \mu_i) \cup \{(p, a_i, q_i) \rightarrow p_{a_i}\} \rangle$.

3.3 Modelling Composite Components

We define Probabilistic Component Automata (PCA) as a probabilistic extension to IA [19] with support for the representation of failures. Probabilistic information is added to the transitions between states and we redefine accordingly the semantics of the operators to construct single and composite models. In composite models we follow the *wait on call* semantics of IA and introduce a different, and arguably more intuitive, normalisation than the one used in PIOA and PCIP. We further introduce an explicit representation for failure actions and failure handling actions that is analogous to the conventional use of exceptions in object-oriented programming languages. Our model has been implemented as an extension to the LTSA tool [1] and is available at *https://wp.doc.ic.ac.uk/dse/software/ltsa-pca/*. The implementation aspects are described in [23].

3.4 Definition

A Probabilistic Component Automaton is defined as $P = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$ where:

- $\mathcal{S}$ is a 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 $\mathcal{E}^{int}$ and $\mathcal{E}^{out}$ are internal actions and output actions, respectively;

- $\Delta \subseteq (\mathcal{S} \times \mathcal{E} \times \mathcal{S})$ is the set of transitions.
- $\mu : \Delta \to [0, 1]$ where $\mu = p(s', a \mid s')$ denotes the probability of reaching state s' from state s through the execution of action a.

Similarly to IA, *input* actions model the receiving end of a communication channel or (interface) methods that can be called, whereas *output* actions model the invocation of methods or sending of messages. Hereafter we refer to both types of interactions as communication between components.

3.5 Modelling Basic Components

Just as Finite State Processes are used to specify Labelled Transitions Systems, Probabilistic Finite State Processes (P-FSP) [23] support incremental specification of PCA models. The correspondence between P-FSP expressions and PCA models is given by the function $pca : E \to \text{PCA}$. 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 2.

$$\frac{}{(a, p_a \to E) \xrightarrow{a, p_a} E} \qquad \text{(Rule 1)}$$

A transition consists of *a)* an action type: ? for input, ! for output, no-symbol for internal, $\sim$ for internal failures, $\sim$? for input failures and $\sim$! for output failures; *b)* the execution probability p, and *c)* the action label $a \in \mathcal{E}$. The corresponding PCA is given by $pca(a, p_a \to E) = \langle \mathcal{S} \cup \{p\},\ p,\ \mathcal{E} \cup \{(p, a, q)\},\ \mu \cup \{(p, a, q) \to p_a\} \rangle$.

$$\frac{}{(p_1\ a_1 \mathrel{->} E_1 \mid \ldots \mid p_n\ a_n \mathrel{->} E_n) \xrightarrow{a_i, p_{a_i}} E_i} \qquad \text{(Rule 2)}$$

The *choice* operator defines possible outcomes from a given state. If $a_1, \ldots, a_n$ are locally controlled actions (internal or output), then $(p_1\ a_1 \mathrel{->} E_1 \mid \ldots \mid p_n\ a_n \mathrel{->} E_n)$ describes a PCA that initially engages in any action a_i with probability p_i. In this case, the choice operator can be used to model `if` or `switch` statements. On the other hand, if $a_1, \ldots, a_n$ are input actions, the action a_i the PCA engages in is dictated by the environment, *i.e.* other processes that output a_i. As input actions follow reactive

semantics, the probabilities associated with transitions with the same action label are equal to 1. Once an input label is chosen, the process can choose locally different outcomes with different probabilities. This case models how the provided interfaces of a component are used by other components, which is only known in a specific architectural configuration i.e., when the components are bound. The corresponding PCA model is formally defined by the following. Let $1 \leq i \leq n$ and $pca(E_i) = \langle \mathcal{S}_i, q_i, \mathcal{E}_i, \Delta_i, \mu_i \rangle$, then $pca(\rho_1\ a_1\ ->\ E_1 \,|\, \ldots \,|\ \rho_n\ a_n\ ->\ E_n) = \langle (\bigcup_i \mathcal{S}_i) \cup \{p\}, p, (\bigcup_i \mathcal{E}_i) \cup \{a_1, \ldots, a_n\}, (\bigcup_i \Delta_i) \cup \{(p, a_i, q_i)\}, (\bigcup_i \mu_i) \cup \{(p, a_i, q_i) \rightarrow p_{a_i}\} \rangle$.

3.6 Modelling Composite Components

While the previous operators enable the specification of basic components, the *parallel composition* operator $||$ is used to automatically construct the PCA model of a composite component from the PCAs representing its subcomponents. The semantics of composite models is a probabilistic extension of the composition semantics of IA models. Synchronisation between input and output actions models the interactions between two components i.e., communication along component bindings and internal actions of different PCAs are interleaved to model their concurrent execution. Note that 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, this implies that parallel composition can only be applied to synchronise single bindings to a provided interface. When there are multiple bindings, the interface actions of the components have to be differentiated before constructing the composite model (see Section 3.7).

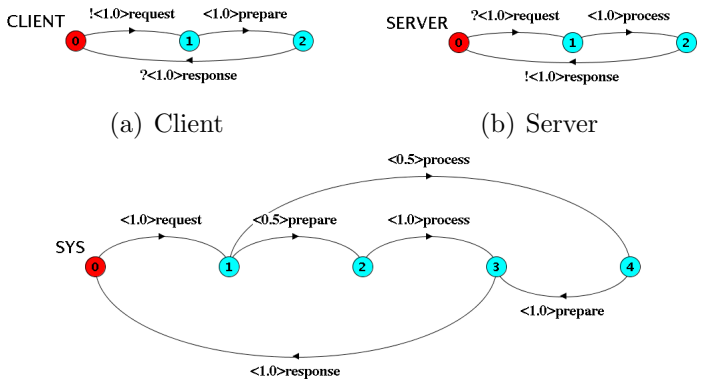
$$\frac{A \xrightarrow{(!a, p_a)} A', B \xrightarrow{(?a, p_{a'})} B'}{A||B \xrightarrow{(a, \frac{p_a \cdot p_{a'}}{\eta})} A'||B'}, a \in \mathcal{E}_A \cup \mathcal{E}_B \qquad \text{(Rule 3)}$$

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

Synchronisation occurs only when both components are ready to communicate, as input actions wait for a corresponding output action to be ready

for execution and output actions wait for a corresponding input action to be ready for communication (Rule 3). Non-shared actions are interleaved (Rule 4) to represent their concurrent execution. In Rules 3 and 4, η denotes a normalisation factor that preserves the generative semantics of locally controlled actions and is discussed in detail in the following paragraphs.

Consider the composite PCA of a simple Client-Server system constructed from the individual PCAs shown in Figure 1. Client and Server execute independently their respective internal actions `prepare` and `process`, between the shared actions `request` and `response`. To preserve the generative semantics for actions `prepare` and `process` in the composite PCA, their probabilities need to be normalised by η. In the general case, η is equal to the sum of the probabilities of all interleaved locally controlled actions for a given state, which implies that both components (*e.g.* Client and Server) are equally likely to execute their actions. Additionally, the normalisation factor η is not used for actions `request` and `response` since the composite model only contains communication between two components (*cf.* sub-section 3.7).



(a) Client (b) Server

(c) ClientServer

Figure 1: PCA representations of a Client and Server system

In the general case, the normalisation factor η is defined as follows. Consider a composite PCA $A_1||...||A_n$. For each composite state $(s_1, ..., s_n) \in \mathcal{S}_{A_1||...||A_n}$, we define $\Delta^{loc}_{A_1||...||A_n}(s_1, ..., s_n)$ as the set of outgoing transitions from state $(s_1, ..., s_n)$ labelled with locally controlled actions. In this case, $\eta = \sum_{i=1}^{n} p_i = \mu_{A_i}(s_i, a, s'_i) \in [1, n]$ denotes the number of automata with locally controlled actions from state $(s_1, ..., s_n)$. For each transition $\langle (s_1, ..., s_n), (a, p'_i), (s'_1, ..., s'_n) \rangle \in \Delta^{loc}_{A_1||...||A_n}(s_1, ..., s_n)$, $p'_i = \frac{p_i}{\eta}, (s_i, a, s'_i) \in \Delta_{A_i}$, $p_i = \mu_{A_i}(s_i, a, s'_i)$. For each transition $\langle (s_1, ..., s_n), (a, p), (s'_1, ..., s'_n) \rangle \in \Delta^{in}_{A_1||...||A_n}(s_1, ..., s_n)$, $\eta = 1$ as normalisation is applied only to local actions.

3.7 Modelling Multiple Bindings

Given that parallel composition only allows synchronisation between a single pair of matching input-output actions, to bind multiple Clients to the Server's provided interface, the Server PCA needs to be extended to handle requests from multiple clients. This is achieved using a *process sharing* operator similar to the one defined for LTS [1]. The interface actions of the Server PCA (`request` and `response`) are substituted by two prefixed actions that represent the interaction with each Client (*e.g.* `c1.request`). The resulting PCA model constructed as `{c1, c2}::Server` is shown in Figure 2. Once more, the probabilities of the new prefixed output actions need to be normalised in order to preserve the generative semantics of locally controlled actions. In this case, η is equal to the number of prefix labels applied by the process sharing operator.

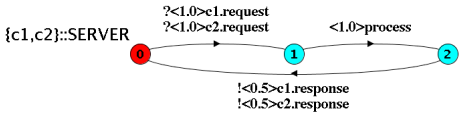


Figure 2: Server PCA with multiple bindings

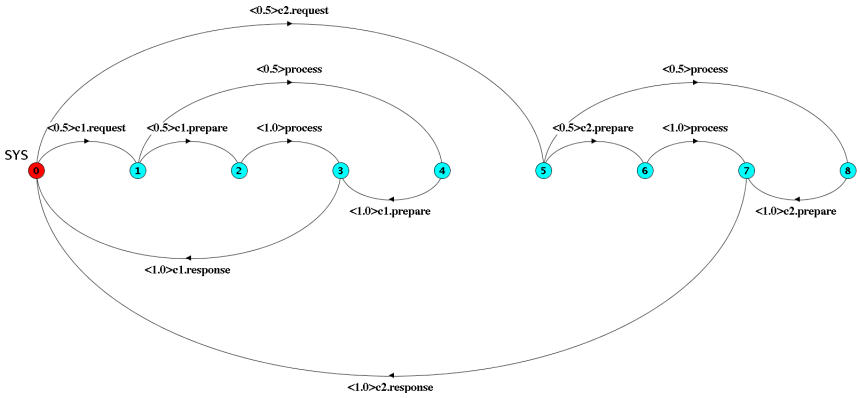


Figure 3: Client-Server composite PCA with multiple bindings

The composite representation of a system comprising two Clients and one Server (Figure 3) can thus be obtained by composing in parallel as follows: `c1:{Client} || c2:{Client} || {c1, c2}::Server`, where `c1:{Client}` adds prefixes `c1` to the actions of the Client PCA. Note that three different kinds of normalisation are applied, each with a slightly different meaning. Firstly, normalisation is applied using the formula defined on the previous

sub-section to the interleaving of requests originating from separate clients (state 0 in Figure 3) where either `c1.request` or `c2.request` can occur. This assumes that the two Clients use the server equally. Secondly, normalisation is applied within the context of the client-server interactions, once for each client (state 1 and state 5 in in Figure 3) where `c1.prepare` or `process` (`c2.prepare` and `process` respectively) can occur. In this case, the normalisation represents that Client and Server are equally likely to execute their actions. Although we have used the same rule in both cases it is important to distinguish between them because normalisation across clients often needs to take into account their differing usage profiles *i.e.* some clients 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. The generalisation of η, to use "weights" for the different clients is straightforward.

Thirdly, the probabilities of transitions labelled with actions `c1.response` and `c2.response` from states 3 and 7 in Figure 3 have to be normalise to reflect that only `c1.response` (`c2.response`) can be executed at state 3 (7) since the Server cannot execute `c1.response` (`c2.response`) without having received `c1.request` (`c2.request`). This normalisation is applied prior to the previous ones as it is applied separately for each machine. In the composition $A_1||...||A_n$, the normalisation factor η_i applied to PCA A_i in the composite state $(s_1, ..., s_n)$ is defined as follows: $\eta_i\langle(s_1, ..., s_n)\rangle = \sum p_j \,|\langle(s_1, ..., s_n), (a, p), (s'_1, ..., s'_n)\rangle \in \Delta_{A_1\|...\|A_n}, a \in \mathcal{E}^{loc}_{A_i}, p_j = \mu_i(s_i, a, s'_i)$. Note that in the case the process sharing has not been previously applied, η_i is equal to 1, hence effectively no normalisation is applied.

The normalisation we use is different from that introduced in PCIP where a single *delay rate* is used to normalise all controlled actions of a composite PCIP. Although the *delay date* can be informally interpreted as the frequency of requests from different components and thus used to normalise requests from different components, it is not suitable for all the aforementioned scenarios, *e.g.* it is not applicable to normalise the probabilities of specific interleaved actions within a Client-Server session.

3.8 Failure Modelling

We introduce failure actions to model failure scenarios, failure propagation and failure handling behaviour. If a PCA is in state s and can execute an unreliable internal action e, a transition $(s, \sim e, \texttt{ERROR})$ leading to the `ERROR` state represents the failure of e. While internal failures represent unexpected executions such as runtime exceptions, transitions labelled with *output failure actions* $(s, \sim!e, \texttt{ERROR})$ 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* $(s, \sim?e, \texttt{ERROR})$ denotes that a PCA is able to *handle* the failure of the corresponding output action from another component. These actions follow reactive semantics as their execution is determined by the PCA that fails.

The semantics of failure propagation and failure handling in PCA is intuitively similar to exception handling. An output failure action can be interpreted as an exception being thrown while an input failure action corresponds to the exception being caught and handled. 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. It can also be handled and a different failure action be output on a different interface *e.g.* to a higher level component.

Figure 4 shows a modified version of the server PCA of our example, where the server's `response` action fails in 1% of occurrences and state -1 denotes the `ERROR` state.

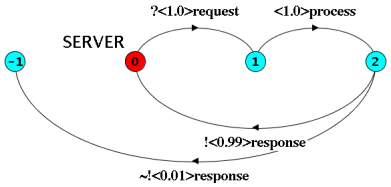


Figure 4: Server PCA with failures

$$\frac{A \xrightarrow{(!a,\, p_a)} A' \,,\; A \xrightarrow{(\sim!a,\, p_f)} \texttt{ERROR} \,,\; B \xrightarrow{(?a,\, p'_a)} B'}{A||B \xrightarrow{(\sim a,\, \frac{p_f \cdot p_a}{\eta})} \texttt{ERROR}} \qquad \text{(Rule 5)}$$

The operational semantics of the parallel composition operator needs to be extended to represent failure propagation and failure handling. When the PCA of the unreliable Server is composed with the Client PCA, the output failure action `response` is *propagated* to the composite PCA as an internal action and the `ERROR` state in the Server PCA becomes a global *ERROR* state (Figure 5). In other words, the failure of a single component, if not handled, leads to the failure of the composite component (Rule 5).

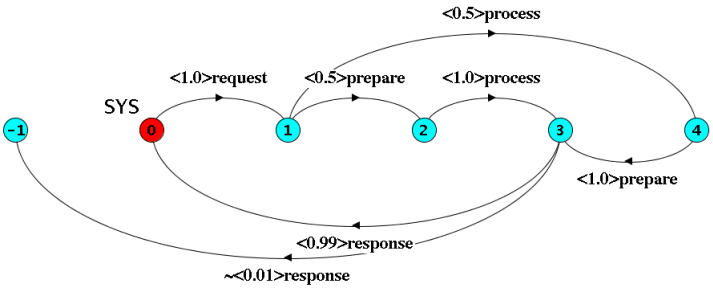


Figure 5: Client-Server composite PCA (with failures)

$$\frac{A \xrightarrow{(!a,\, p_a)} A' \,,\; A \xrightarrow{(\sim !a,\, p_f)} ERROR \,, \quad B \xrightarrow{(?a,\, p'_a)} B' \,,\; B \xrightarrow{(\sim ?a,\, p_{f'})} B''}{A||B \xrightarrow{(a,\, \frac{p_f \cdot p_{f'}}{\eta})} reset(A)||B''} \qquad \text{(Rule 6)}$$

Alternatively, the Client PCA can be extended to handle the failure of the *response* action (Figure 6) using an input failure action followed by failure handling behaviour (Rule 6). 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.

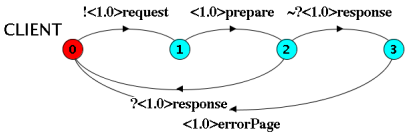


Figure 6: Client PCA with failure handling

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

3.9 Hiding

The composite model for a complex system often is significantly large as it contains a large number of internal states and transitions. The hiding operator $\backslash\{a_1, \ldots, a_n\}$ when applied to a PCA A collapses, when possible, the

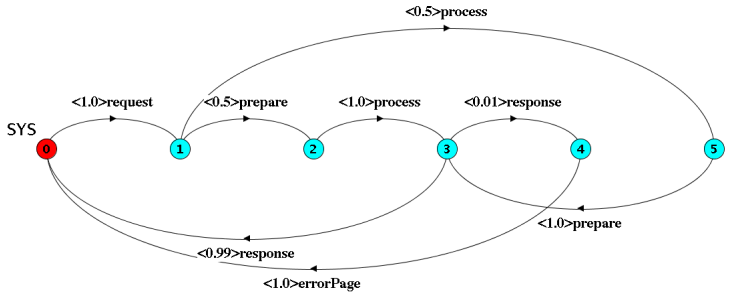


Figure 7: Client-Server composite PCA with failure handling

transitions in A labelled with the action names $\{a_1, \ldots, a_n\}$, while maintaining the probabilistic reachability properties of the original process. This operator can be used to remove behaviour associated with unbound provided interfaces, internal transitions or reduce a PCA to its interface behaviour representation. Its dual, the interface operator @ $\{a_1, \ldots, a_n\}$, indicates the transitions that should be kept and is internally converted to $\setminus\left\{\mathcal{E}_A - \{a_1, \ldots, a_n\}\right\}$. In the next section we present the algorithm we use to implement hiding and minimisation to help reduce the state space and facilitate analysis. The results of this reduction will be discussed in Section 5.

4 Compositional Reliability Analysis

Compositional Reachability Analysis (CRA) [24] has been proposed to help mitigate the state-explosion associated with the composition of LTS and thereby potentially improving the scalability of their composition [17], but so far the method has not been applied to probabilistic systems. Consider two LTS $A = \langle \mathcal{S}_A, q_A, \mathcal{E}_A, \Delta_A \rangle$ and $B = \langle \mathcal{S}_B, q_B, \mathcal{E}_B, \Delta_B \rangle$ representing the behaviour of two components. In the composite behaviour $A||B$, A imposes behaviour contextual constraints on B [24]. Such constraints are captured by an interface process I such that the properties which hold for $A||I$ also hold for $A||B$. This requires that I is behaviourally equivalent to $B \uparrow \alpha A$, *i.e.* a process that is constructed by restricting B to actions in A. The main steps for constructing the interface process I of B constrained by A are as follows:

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

Consider now the PCA representation of two components A and B. If the provided interfaces of B are bound to required interfaces of A, then the PCA representing the behaviour of B can be reduced using the CRA method to its interface actions w.r.t. interactions with component A. However, the CRA method [24] does not take probabilities into account and these need to be propagated when transitions are deleted.

A compositional analysis algorithm which seeks to apply CRA concepts to PIOA has been proposed before [25], but its resulting automaton may be inconsistent with PIOA semantics. For example, transitions labelled with a non-observable action that lead to an absorbing state are deleted [25] but this is not appropriate since their probabilities cannot be propagated to other transitions.

We propose here a novel algorithm to reduce PCA models to their interface behaviour through CRA, whilst preserving the probabilistic properties of the original model. The algorithm analyses the transitions of each state and checks if non observable transitions can be deleted and whether their probabilities can be propagated *forwards* to subsequent transitions or *backwards* to prior transitions.

We define as follows:

- $\Delta(s) = \{(s, e, s') \in \Delta \,|\, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the successor transitions of a state $s \in S$;
- $\rho(s) = \{(s', e, s) \in \Delta \,|\, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the predecessor transitions of a state $s \in S$;
- $\Delta_s(s) = \{s' \,|\, (s, e, s') \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the states of successor transitions of a state $s \in \mathcal{S}$;
- $\rho_s(s) = \{s' \,|\, (s', e, s) \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: states of predecessor transitions of a state $s \in \mathcal{S}$;
- $\Delta_e(s) = \{e \,|\, (s, e, s') \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the actions successor transitions of a state $s \in \mathcal{S}$;
- $\rho_e(s) = \{e \,|\, (s', e, s) \in \Delta, s' \in \mathcal{S}, e \in \mathcal{E}\}$: the actions predecessor transitions of a state $s \in \mathcal{S}$;
- $canFail(e, s) = \exists (s, \sim e, s') \in \Delta$: a non-reliable action $e \in \mathcal{E}$ from state s.

Our algorithm is divided in two phases (Algorithm 1). During the first phase (Algorithm 2), we perform a breadth-first search to delete the following situations:

input : $B = \langle \mathcal{S}, q, E, \Delta, \mu \rangle$ and $\mathcal{E}_A$
output: $I_{A||B} = \langle \mathcal{S}_I, q_I, \mathcal{E}_I, \Delta_I, \mu_I \rangle$

1 *[B', markedStates, cyclicPaths] = firstPhase(B, $\mathcal{E}_A$)*;
2 $I_{A||B}$ = *secondPhase(B', markedStates, cyclicPaths,* $\mathcal{E}_A$);

Algorithm 1: PCA reduction algorithm

- paths initiated by transitions labelled with an unused input action that does not belong to the action set $\mathcal{E}_A$, *e.g.* input actions associated with unbound provided interfaces (Algorithm 2 - line 18);
- transitions labelled with a locally controlled action that does not belong to $\mathcal{E}_A$ and that verify one of the following conditions: *a*) the destination state has only one incoming transition and does not have transitions labelled with input actions (Algorithm 2 - line 26) and *b*) the transition is the only outgoing transition from the current state and there is no incoming transition labelled with input actions (Algorithm 2 - line 29);

The following transitions are kept in order to preserve the semantics of PCA models:

- cyclic transitions, *i.e.* $(s, e, s') \in \Delta, s = s'$ (Algorithm 2 - line 15);
- transitions labelled with locally controlled actions which: *a*) lead to a deadlock or error state (Algorithm 2 - line 20) and *b*) can fail (Algorithm 2 - line 23).

Additionally, the number of cyclic paths that start at a given state are measured in the first phase (Algorithm 2 - line 8) to be later used in the second phase (Algorithm 3). Transitions leading to a state s that has several incoming transitions are kept in the first phase and the state s is marked for subsequent analysis in the second phase.

In the second phase, the reduced automaton produced by the first phase is traversed and the incoming transitions of all the marked states are grouped based on their source state. For each marked state s, $\rho(s, s')$ denotes the set of incoming transitions of state s originating from s' ($s' \in \rho_s(s)$). Transitions in $\rho(s, s')$ are collapsed backwards if all the transitions are labelled with actions that do not belong to $\mathcal{E}_A$ and no transition in the group is labelled with an input action. In the specific case when $|\rho_s(s)| = 1$, *i.e.* all incoming transitions of marked state s have the same source state, these transitions are collapsed forward if the previous conditions are verified.

A breadth first navigation is used to traverse the automaton, though a state is only analysed when all its incoming transitions have been traversed

```
input  : B = ⟨S, q, E, Δ, μ, δ⟩ and E_A
output: B' = ⟨S', q, E', Δ', μ', δ'⟩, markedStates, cyclicPaths
1  boolean[] markedStates, boolean[] visited;
2  int[] cyclicPaths; Queue states;
3  states.push(q);
4  while not states.isEmpty() do
5      currentState ← states.pop();
6      if visited[currentState] then
7          cyclicPaths[currentState]++;
8          continue;
9      visited[currentState] = true;
10     if Δ(currentState) = ∅ then continue;
11     ;
12     foreach (currentState, e, s) ∈ Δ(currentState) do  if e ∈ E_A
       then
13         addTransition((currentState, e, s)) to B';
14     else
15         if currentState = s then
16             addTransition((currentState, e, s)) to B';
17             continue;
18         if e ∈ E_in then  continue;
19         ;
20         if Δ(s) = ∅ ∧ e ∉ E^in then
21             addTransition((currentState, e, s)) to B';
22         else if |ρ(s)| = 1 then
23             if canFail(e, currentState) then
24                 addTransition((currentState, e, s)) to B';
25             else if Δ_e(s) ∩ E^in = ∅ ∧ s ≠ q then
26                 collapseForward (currentState, s, μ(currentState,
                   e, s));
27             else if
               |Δ(currentState)| = 1 ∧ ρ_e(currentState) ∩ E^in = ∅ then
28                 collapseBackward(currentState, s, μ(currentState,
                   e, s));
29             else
30                 addTransition((currentState, e, s)) to B;
31         markedStates[s] = true;
32     if s ≠ q ∧ s ≠ currentState then  states.push(s);
33     ;
34 ;
```

Algorithm 2: First Phase

input : $B' = \langle \mathcal{S}', q, \mathcal{E}', \Delta', \mu', \delta' \rangle$, boolean[] markedStates, int[] cyclicPaths
output: $I_{A||B} = \langle \mathcal{S}_I, q_I, \mathcal{E}_I, \Delta_I, \mu_I, \delta_I \rangle$ and $\mathcal{E}_A$

```
1  int[] nTimesVisited;
2  Queue states;
3  I_{A||B} = clone(B')
4  states.push(q_I);
5  while not states.isEmpty() do
6      currentState ← states.pop();
7      if markedState[currentState] then
8          ρ_agg ← ρ(currentState) indexed by source state s from I_{A||B};
9          μ_agg ← aggregated reachability of ρ(currentState) by source
           state s from I_{A||B};
10         if |ρ_agg| = 1 then
11             s ← single predecessor state;
12             if ρ_{e-agg} \ E_A = ∅ ∧ s ≠ currentState then
13                 if Δ'(currentState) \ E^in = ∅ then
14                     collapseForward (s, currentState, μ_agg(s));
15                 else
16                     if ρ_e(s) \ E^in = ∅ then
17                         collapseBackward (μ_agg(s), s);
18         else
19             foreach s ∈ index(ρ_agg) do
20             if ρ_{e-agg}(s) \ E_A = ∅ ∧ ρ_e(s) \ E^in = ∅ ∧ s ≠ currentState
               then
21                 collapseBackward (μ_agg(s), s);
22             ;
23     foreach (currentState, e, s) ∈ Δ(currentState) do if
       currentState ≠ s then nVisits[s]++ ;
24     if s ≠ q ∧ nVisits[s] = ρ(s) − cyclicPaths[s] then
       states.push(s) ;
25     ;
```

Algorithm 3: Second phase

1 **foreach** $(s', e', s'') \in \Delta(s')$ **do**
2 *change* (s', e', s'') *to* $(s,\ e', s'')$;
3 $\mu(s, e', s'') = \mu(s', e', s'') \times \mu_{cf}$;

Algorithm 4: collapseForward(s, s', μ_{cf})

1 **foreach** $(s'', e, s) \in \rho(s)$ **do**
2 *change* (s'', e, s) *to* (s'', e, s');
3 $\mu'(s'', e, s') = \mu'(s'', e, s) \times \mu_{cb}$;

Algorithm 5: collapseBackward(s, s', μ_{cb})

(Algorithm 3 - line 24). Transitions that represent the end of a cyclic path which starts and ends in state s are not considered when verifying if all incoming transitions have been traversed. While the former condition ensures that all possible reductions have been applied when the incoming transitions of a marked state are considered, the latter condition allows the automaton navigation to progress.

Although the result of the two phases is an automaton that cannot be reduced further, it may be necessary to keep transitions beyond those in the set $\mathcal{E}_A$ given by the user, when the probabilities cannot be meaningfully propagated to other transitions (forwards or backwards).

We are currently working on a formal proof to show that the reduced model produced by the algorithm bisimulates the original model. Thus far we have thoroughly tested the algorithm and its implementation in LTSA-PCA to check that the reduced model preserves the probabilistic characteristics of the original across an extensive set of examples.

5 Evaluation

We have implemented an extension of the LTSA tool [26] to support the specification of Probabilistic Component Automata using a modified version of FSP (PCA-FSP) that considers input, internal, output and failure actions as well as probabilistic information associated with those actions. PCA are then constructed from specifications in PCA-FSP using the operational semantics presented in section 3. A closed PCA, *i.e.* one without input-actions, is then automatically translated to an equivalent DTMC specification to be analysed in PRISM. The extended version of the LTSA tool and the examples used in this paper can be found at `http://www.doc.ic.ac.uk/~pr1810/`.

An evaluation of the complexity gains of reliability analysis when using the reduction algorithm to compute the overall system representation is shown in Table 1. 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 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 in the extended LTSA tool and the the time it takes to analyse the system reliability in PRISM. *No Reduct.* denotes the time to compute the composite representation without using the reduction algorithm, while *Reduct.* represents the time to reduce the component representations and then to construct the composite system model using the reduced representations. The analysis time obtained from PRISM includes the time to construct the internal representation in PRISM from the DTMC specification generated by LTSA, as well as the time to analyse the overall system reliability. 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 Air with the following specifications: 1.8 GHz Intel Core i7, 4GB 1333 MHz, 256GB SSD.

	Original Size		**Reduced Size**		**Execution Time**			
Name	States	Transitions	States	Transitions	No Reduct.	Analysis	Reduct.	Analysis
Web-Server (2 Clients)	100	216	8 (92%)	18 (92%)	0.047s	0.105s	0.046s (1%)	0.007s (93%)
Web-Server (3 Clients)	296	792	20 (93%)	60 (92%)	0.048s	0.546s	0.046s (4%)	0.017s (97%)
Web-Server (4 Clients)	784	2496	48 (95%)	176 (94%)	0.053s	4.11s	0.047s (11%)	0.019s (99.5%)
Web-Server (5 Clients)	1952	7200	112 (96%)	480 (96%)	0.072s	34.57s	0.052s (27%)	0.082s (99.7%)
Web-Server (6 Clients)	4672	19584	256 (96%)	1248 (96%)	0.125s	294.82s	0.054s (56%)	0.082s (99.97%)
Web-Server2 (2 Clients)	488	1262	34 (96%)	84 (96%)	0.06s	1.13s	0.052s (13%)	0.13s (88%)
Web-Server2 (3 Clients)	4336	14184	184 (96%)	636 (96%)	0.125s	197.5s	0.055s (56%)	0.255s (99.8%)
Web-Server2 (4 Clients)	33334	123929	2250 (96%)	11250 (96%)	1.9s	¿ 600s	0.11s (94%)	1.12s (¿ 99.99%)

Table 1: Reduction Algorithm Evaluation Results

We have analysed in this manner two Client-Server systems with increasing numbers of clients. In the first system, the Server is a composite component that includes a Web-Server which handles requests from Clients and interacts with backend components (a Cache and a Database) to get the requested content. The second system (Web-Server2) includes a backup server that is used by the Clients when the main Server fails to send the requested content. Table 1 reports results for the Client-Server system using from 2 to 5 Clients and one Server. 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 as expected. For smaller models (Tele Health systems

and Client-Server with 2 and 3 clients) applying the reduction algorithm and then composing requires broadly the same 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 11%, 27% 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 99.96% 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 representation from the DTMC specification of the Client-Server system. These results show the usefulness of the reduction algorithm as an effective technique for scalable reliability analysis of component-based systems.

6 Conclusion and Future Work

Our Probabilistic Component Automata provide an expressive formalism to model the probabilistic failure behaviour of components. By combining the semantics of generative and reactive models it is possible to construct composite probabilistic behaviour representations that cater for reusability, failure scenarios and how these are handled. The specification burden is further reduced by modelling each component individually, thereby facilitating incremental elaboration and enabling the definition of more fine grained representations. A close correspondence with the source code is also supported, thus producing a faithful representation of the implementation.

Although the analysis itself is performed using a closed DMTC, the composite system representation is automatically constructed from the representations of its parts. Models of individual components can be fine grained to allow detailed analysis of the execution profile. However, when analysing overall properties of systems the same level of detail may not be required, in particular for system reconfiguration where components are replaced as a whole. Using hiding and minimisation, smaller composite models can be con-

structed by reducing the representations of sub-components, before applying parallel composition. The reduction gains can be significant but depend on the properties analysed, as they determine which internal actions can be removed.

Another advantage of constructing the system representation in this way is that third-party providers can automatically generate and provide interface behaviour representations of their components without having to disclose internal behaviour. PCA establishes here a close correspondence between behavioural and architectural aspects. Input and output actions correspond to provided and required interfaces of component models such as Darwin [6]; the architectural structure of composite components, which hides internal bindings between sub-components, is preserved at the behaviour level by applying the reduction algorithm to compute their interface behaviour.

We intend to extend PCA, its operators and the reduction algorithm with variables to support late specification of transition probabilities. Such *parametric PCA models* would allow re-analysing reliability properties after changes in the execution profile without having to re-construct the composite model and re-run the model checking tools (*cf.* [27] for analysis with parametric DTMC models). These would provide means for scalable, accurate probabilistic analysis at runtime. We also plan to investigate the use of PCA along with the reduction algorithm to perform reliability analysis of alternative configurations for runtime adaptation.

Acknowledgements

This work was supported by Fundação para a Ciência e Tecnologia under the grant SFRH/BD/73967/2010.

References

[1] J. Magee and J. Kramer, *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[2] 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.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[3] K.-C. Tai and P. V. Koppol, "An incremental approach to reachability analysis of distributed programs," in *Proceedings of the 7th international workshop on Software specification and design*, ser. IWSSD '93. Los Alamitos, CA, USA: IEEE Computer Society Press, 1993, pp. 141–150.

[4] W. J. Yeh and M. Young, "Compositional reachability analysis using process algebra," in *Proceedings of the symposium on Testing, analysis, and verification*, ser. TAV4. New York, NY, USA: ACM, 1991, pp. 49–59.

[5] D. Garlan, R. T. Monroe, and D. Wile, "Foundations of component-based systems," G. T. Leavens and M. Sitaraman, Eds., 2000, ch. Acme: architectural description of component-based systems, pp. 47–67.

[6] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, "Specifying distributed software architectures," in *ESEC*. London, UK: Springer-Verlag, 1995.

[7] N. Medvidovic and R. N. Taylor, "Software architecture: foundations, theory, and practice," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 471–472. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810435

[8] J. Hillston, *A compositional approach to performance modelling*. New York, NY, USA: Cambridge University Press, 1996.

[9] R. Cheung, "A user-oriented software reliability model," *Software Engineering, IEEE Transactions on*, vol. SE-6, no. 2, pp. 118 – 125, march 1980.

[10] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, "Reliability analysis of component-based systems with multiple failure modes," in *Proceedings of the 13th international conference on Component-Based Software*

Engineering, ser. CBSE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 1–20.

[11] W.-L. Wang, D. Pan, and M.-H. Chen, "Architecture-based software reliability modeling," *Journal of Systems and Software*, vol. 79, no. 1, pp. 132 – 146, 2006.

[12] L. Cheung, R. Roshandel, N. Medvidovic, and L. Golubchik, "Early prediction of software component reliability," in *ICSE '08.*, may 2008, pp. 111 –120.

[13] G. Rodrigues, D. Rosenblum, and S. Uchitel, "Using scenarios to predict the reliability of concurrent component-based software systems," in *FASE'05*. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 111–126.

[14] A. Sokolova and E. P. D. Vink, "Probabilistic automata: System types, parallel composition and comparison," in *In Validation of Stochastic Systems: A Guide to Current Research*. Springer, 2004, pp. 1–43.

[15] S.-H. Wu, S. A. Smolka, and E. W. Stark, "Composition and behaviors of probabilistic i/o automata," in *Theoretical Computer Science*, 1994, pp. 513–528.

[16] I. Krka, L. Golubchik, and N. 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*, ser. QUOVADIS '10. New York, NY, USA: ACM, 2010, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/1808877.1808881

[17] S. Graf and B. Steffen, "Compositional minimization of finite state systems," in *In Proc. 2ND International Conference of Computer-Aided Verification*, 1991, pp. 186–196.

[18] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 137–151. [Online]. Available: http://doi.acm.org/10.1145/41840.41852

[19] L. de Alfaro and T. 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*, ser. ESEC/FSE-9. New

York, NY, USA: ACM, 2001, pp. 109–120. [Online]. Available: http://doi.acm.org/10.1145/503209.503226

[20] E. Pavese, V. Braberman, and S. Uchitel, "Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 335–344. [Online]. Available: http://doi.acm.org/10.1145/1595696.1595760

[21] S. S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," *Dependable and Secure Computing, IEEE Transactions on*, vol. 4, no. 1, pp. 32 –40, jan.-march 2007.

[22] I. Krka, G. Edwards, L. Cheung, L. Golubchik, and N. Medvidovic, "A comprehensive exploration of challenges in architecture-based reliability estimation," in *WADS*, 2008, pp. 202–227.

[23] P. Rodrigues, E. Lupu, and J. Kramer, "Ltsa-pca: Tool support for compositional reliability analysis," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 548–551. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591067

[24] S. C. Cheung and J. Kramer, "Context constraints for compositional reachability analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 334–377, Oct. 1996.

[25] E. W. Stark and G. 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)*, 1999, pp. 3–24.

[26] J. Magee, "Behavioral analysis of software architectures using ltsa," in *Proc. of the 21st Int. Conf. on Software engineering*, ser. ICSE '99, 1999, pp. 634–637.

[27] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 341–350. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985840