Compositional Reliability Analysis for Probabilistic Component Automata

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

***Abstract*—In this paper we propose a modelling formalism, Probabilistic Component Automata (PCA), as a probabilistic extension to Interface Automata to represent the probabilistic behaviour of component-based systems. The aim is to support composition of component-based models for both behaviour and non-functional properties such as reliability. We show how additional primitives for modelling failure scenarios, failure handling and failure propagation, as well as other algebraic operators, can be combined with models of the system architecture to automatically construct a system model by composing models of its subcomponents. The approach is supported by the tool LTSA-PCA, an extension of LTSA, which generates a composite DTMC model. The reliability of a particular system configuration can then be automatically analysed based on the corresponding composite model using the PRISM model checker. This approach facilitates configurability and adaptation in which the software configuration of components and the associated composition of component models are changed at run time.**

I. Introduction

In component-based systems, it is often beneficial to specify both behaviour and non-functional properties at the component level. In building systems as a software architecture of components, composition can be used to automatically compose and model both the behaviour and non-functional properties of that system. In general, we advocate models of non-functional properties which *a)* are compositional to support automatic non-functional analysis of a given configuration from the models of each component; *b)* define a link with architectural configuration as the non-functional properties of a component also depend on the context in which it is deployed, *i.e.* bindings; *c)* have suitable semantics in order to accurately represent the behaviour of software components.

Although this has been successfully explored for functional behaviour models using LTS representations [1], [2], [3], [4], these models do not include stochastic information, *e.g.* time. Probabilistic extensions of LTS models [5], [6], [7] model frequency of execution of actions and or their duration but do not accurately capture the behaviour of component-based systems due to the difficulty of representing probabilistic behaviour [8]. Both LTS and the derived extensions do not take into account the architectural configuration of a system [9], [10], [11] when constructing the composite model. As a consequence, they implicitly assume that all the provided functionality of a component is used in any configuration. Moreover, existing approaches for non-functional properties that combine behaviour models with architectural configuration are based upon manual representations of the control-flow between components of a specific configuration [12], [13].

In this paper we propose a modelling formalism, Probabilistic Component Automata (PCA) which complements complements Architectural Description Languages, such as Darwin [10], to construct the system behaviour model of a given system configuration from the representation of its parts. Our model includes primitives to represent failure scenarios, failure propagation and failure handling which closely resemble how exceptions are used to deal with failures in object-oriented languages. Before describing PCA in Section III, we first conduct a brief review in Section II of existing formalisms for modelling probabilistic behaviour, their inter-relationships and limitations. We then illustrate the application of PCA for reliability analysis using an example client-server system in Section IV. Finally, we present conclusions and future work in Section V.

II. Related Work

A. Non-composable models

Discrete-Time Markov Chains (DTMCs) have been initially proposed by Cheung [14] to represent the reliability of component-based systems. 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. Existing extensions to cater for failure propagation [15] and to consider mappings between architectural patterns and corresponding DTMC representations [16] also need be manually defined and as the model for the entire system cannot be automatically constructed from the representation of its parts.

B. Composable Models

Although performance models are compositional [17], 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?".

Probabilistic I/O Automata (PIOA) [5] are a probabilistic extension of I/O Automata [18] that distinguish between *input actions*, in which execution is determined by the *environment*, and *internal/output actions* which are controlled by the component. 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. Although distinguishing between input and output actions allows PIOA to address some of the inconsistencies encountered in probabilistic LTS [19], PIOA are required to be *input-enabled*, *i.e.* each component must process any input action at any time, regardless of its internal state. This makes PIOA unsuitable for representing component-based software systems.

Probabilistic Component Interface Protocols (PCIP) [6] are a full probabilistic extension[1] to Interface Automata, which also makes a distinction between input/output and internal and do not require input-enabledness. While in Interface Automata input/output actions wait for the corresponding output/input action to be ready for interaction, in PCIP a transition leading to a special *error state* is included in the composite model whenever an output action is ready to be executed and cannot immediately synchronise with the corresponding input action. These semantics hinder the applicability of PCIP for modelling the behaviour of software components as method invocations, denoted by output actions, are synchronous. Note that asynchronous method calls can be modelled with synchronous interactions using an additional component. On top of that, traditional reliability analysis is based upon transitions to the error state that 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 [20], [21]. However, a compositional model that establishes a link with architectural models and allows for representation for failures, failure propagation and failure handling is still missing.

III. Probabilistic Component Automata

We define Probabilistic Component Automata (PCA) as a probabilistic extension to IA [22]. Probabilistic information is added to the transitions between states and we redefine accordingly the semantics of the operators to construct single and composite models which can be related to programming primitives. In composite models we follow the synchronisation semantics of IA as it is closer to the behaviour of software components. 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 (OO) programming languages. In addition, we show how architectural information is necessary to construct the composite model corresponding to a given system configuration.

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].

A. 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.
- $\mu : \Delta \rightarrow [0, 1]$ where $\mu(s, a, s')$ denotes the probability of reaching state s' from state s through the execution of action a, subject to:

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

$$\underbrace{\forall s \in \mathcal{S}, \forall a \in \mathcal{E}^{in}, \Big(\sum_{(s,a,s') \in \Delta} \mu(s, a, s') \Big) = 1}_{\text{reactive semantics}} \tag{2}$$

B. 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 determined by the function $pca : E \rightarrow$ PCA. Given a P-FSP expression E, $pca(E) = \langle \mathcal{S}, q, \mathcal{E}, \Delta, \mu \rangle$.

Prefix and choice are the basic P-FSP operators to incrementally construct PCA models of basic components, whose operational semantics are respectively defined by Rules 1 and 2. The operational semantics rules should be read as $\frac{Hypothesis}{Conclusion}$.

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

A transition denotes a single instruction and consists of *a)* the execution probability p, *b)* the action label $a \in \mathcal{E}$ and *c)* an action type:

- ? for input actions that model the receiving end of a communication channel or methods that can be called;
- ! for output actions which denote the invocation of methods or sending of messages;
- no-symbol for internal actions;
- $\sim$ for internal failures, $\sim$? for input failures and $\sim$! for output failures for failure modelling (more details in the next sub-section).

[1] Probabilistic Interface Automata [7] only support the composition of a probabilistic model of the environment with a non-probabilistic model of the software system.

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

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

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. Input actions are given a default probability of 1.0 as, on composition, these actions inherit the probability form the corresponding output action of the connected component 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. In both cases, 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 $pca(\rho_1\ a_1\ {-\!>}\ E_1 \,|\, \ldots \,|\ \rho_n\ a_n\ {-\!>}\ E_n) = \langle (\bigcup_i \mathcal{S}_i) \cup \{p\},\ p, (\bigcup_i \mathcal{E}_i) \cup \{a_1, \ldots, a_n\}, (\bigcup_i \Delta_i) \cup \{(p, a_i, q_i)\},\ (\bigcup_i \mu_i) \cup \{(p, a_i, q_i) \rightarrow p_{a_i}\} \rangle$.

C. Modelling Composite Components

While the previous operators enable the specification of basic components, the *parallel composition* operator $||$ is used to construct the PCA model of a composite component from the PCAs representing its sub-components. 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:

$$E_A^{int} \cap E_B = \emptyset, E_B^{int} \cap E_A = \emptyset,$$
$$E_A^{in} \cap E_B^{in} = \emptyset, E_A^{out} \cap E_B^{out} = \emptyset.$$

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 a configuration includes multiple bindings, the interface actions of the components involved have to be differentiated before constructing the composite model (see sub-section III-E).

$$\frac{A \xrightarrow{(!a, p_a)} A',\ B \xrightarrow{(?a, p_{a'})} B'}{A||B \xrightarrow{(a, \frac{p_a \cdot p_{a'}}{\eta})} A'||B'} \quad \text{(Rule 3)}$$

$$\frac{A \xrightarrow{(?a, p_a)} A',\ B \xrightarrow{(!a, p_{a'})} B'}{A||B \xrightarrow{(a, \frac{p_a \cdot p_{a'}}{\eta})} A'||B'} \quad \text{(Rule 3)}$$

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

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). Internal actions are interleaved (Rule 4) to denote their concurrent execution. In Rules 3 and 4, η denotes a normalisation factor that preserves the generative semantics of locally controlled actions. For instance, consider that two PCA are both ready to execute internal actions x and y and no other actions can be executed from their current state; their concurrent execution is represented as two execution paths: $a \rightarrow b$ and $b \rightarrow a$. The normalisation factor determines the probability of each of first action of each of those paths. A full description of η and the associated cases is given in [24].

D. 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 (see equation 1). 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 (see equation 2).

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 the representation of 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.

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

The operational semantics of the parallel composition operator needs to be extended to represent failure propagation and failure handling. Rule 5 defines that the failure of a single component, if not handled, leads to the failure of the composite component. If the failure action is handled, then it

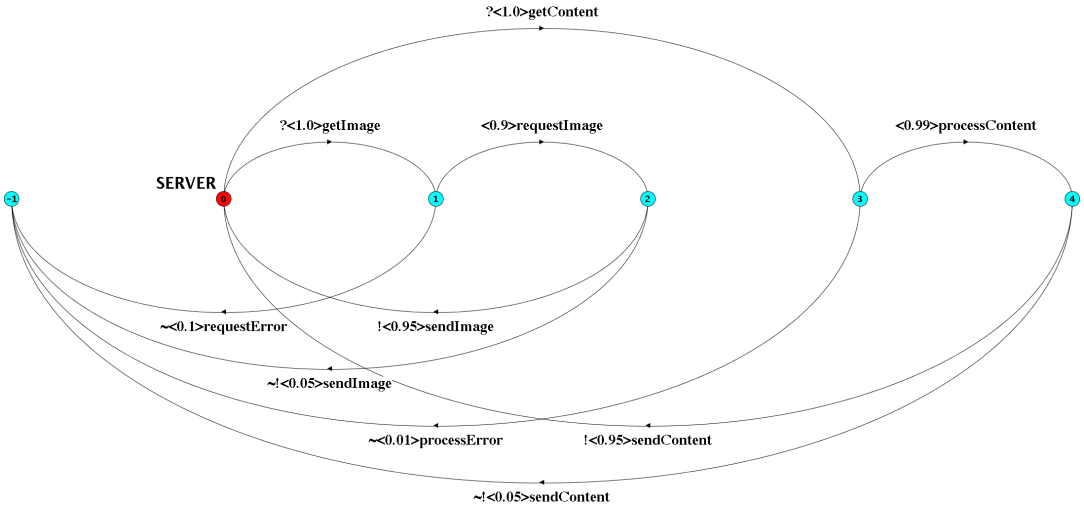

Fig. 1. PCA model of Server component

becomes a local action in the composite model (Rule 6). While the component that handles the failure executes the failure handling behaviour defined by B'', the component that raises the exception resets its behaviour to the initial state to be able to continue further interactions.

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

E. Relabelling and Hiding

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

Additionally, when applied to a PCA A, the hiding operator $\backslash\{a_1, \ldots, a_n\}$ collapses, when possible, the transitions in A labelled with the internal or input actions $\{a_1, \ldots, a_n\} \subseteq \mathcal{E}^{int} \cup \mathcal{E}^{in}$, while maintaining the probabilistic reachability properties of the original process. When applied to input actions, this operator removes behaviour associated with unbound provided interfaces, while when used for removing internal transitions it reduces a PCA to its interface behaviour representation. A full description of the algorithm that implements the hiding operator and the complexity gains in compositional reliability analysis are given in [24].

F. Automatic Construction of System model

After describing the semantics of the operators supported by PCA, we now discuss how these can be used in conjunction with the architectural configuration of a system to automatically construct its composite representation. Consider an architectural configuration $\mathcal{B}_{arch}$ defined based on bindings between provided and required interfaces of a set of components. Before constructing the corresponding composite PCA model of a given architectural configuration, the following steps are applied to the PCA A_{C_i} model of each component C_i.

1) Remove behaviour associated with unbound provided interfaces:
   - given set of unbound interfaces, determine sub-set of input actions associated with unbound interfaces and compute PCA $A_{C_i - \mathcal{B}_{arch}}$ denoting *active* behaviour w.r.t. $\mathcal{B}_{arch}$;
2) Compute interface representation $IA_{C_i - \mathcal{B}_{arch}}$ by removing internal behaviour;

In case there are multiple bindings to a provided interface in $\mathcal{B}_{arch}$ the relabelling operator is applied both to the actions associated with the multiple required interfaces and the provided interface in order to distinguish requests from the different components. After the above steps have been applied, the composite model corresponding to the system configuration $\mathcal{B}_{arch}$ is obtained by composing the interface representation of each component:

$$A_{\mathcal{B}_{arch}} = IA_{C_1 - \mathcal{B}_{arch}} \parallel \ldots \parallel IA_{C_n - \mathcal{B}_{arch}}.$$

Given that each interface representation $IA_{C_1 - \mathcal{B}_{arch}}$ does not contain behaviour associated with unbound provided interfaces, the composite model $A_{\mathcal{B}_{arch}}$ 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 can be used in PRISM model checker for analysis of reliability properties [23].

IV. Example

In this section we describe a simple Client-Server system to illustrate how the composite model for a given system

configuration is constructed using the PCA model of each component and a specific architectural configuration.

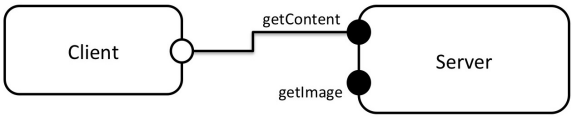

Fig. 2. Architectural configuration of an example Client-Server system

Consider the PCA models of the Client (Figure 3) and the Server (Figure 1) components. The Client PCA starts with

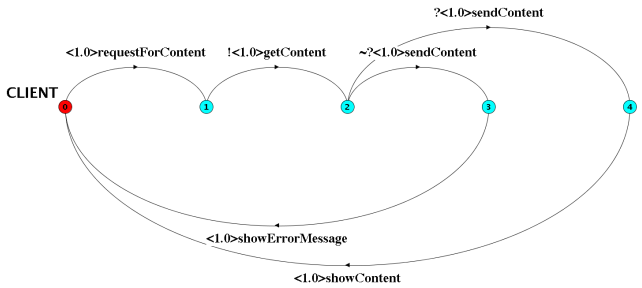

Fig. 3. PCA model of Client component

an internal action denoting a request, which then leads to an output action representing the invocation of an interface $getContent$. In order to accommodate both method invocations and message passing systems, an output action does not wait for the result to be returned. Consequently, the output action $getContent$ is immediately followed by an input action $sendContent$ to denote the result of interface $getContent$ has been successfully received. On the other hand, the input failure action $getContent$ allows the representation of failure handling behaviour in the case that the Client does not receive a result. The internal action $showErrorMessage$ can be seen as the instructions included in a *catch* block for OO languages.

Furthermore, the Server PCA model starts with two input actions denoting that the Server can respond to an invocation to any of its two provided interfaces. After invocation, both interfaces execute an internal action $requestImage/processContent$ to address the request, which may fail with a certain probability; if the request is successfully processed, a result is sent through a channel which is 95% reliable.

Note that these models are not dependent on a particular architectural configuration and can be used in any context in which the components are deployed. For instance, both the PCA of the Server and Client components do not assume that will interact with a particular Client/Server. However, both Server and Client assume a certain ordering of input/output actions for interactions through bindings. In fact, a specific architectural configuration can be valid, w.r.t. to compatible bindings between provided and required interfaces, whilst a mismatch between the expected interaction protocols of both components exists. Although the Client and Server components have a compatible interface protocol, the composite model corresponding to the architectural configuration in Figure 2 could be used to test for interface behaviour compatibility using standard behavioural model checking techniques such as deadlock analysis [1]. Such analysis can be used in architectural assembly processes to filter incompatible components, thus preventing the system from halting at runtime as incompatible components are not deployed.

Furthemore, given that the Client is only bound to the $getContent$ interface of the Server, the Server PCA needs to be adjusted by removing the behaviour associated with unbound interface $getImage$. We also remove the internal actions from both the modified Server PCA and the Client PCA and then compute the composite model (Figure 4). The resulting individual PCA models before composition only have input/output actions and internal failure actions, though the latter can be relabelled in order to omit all the information about internal behaviour. As a result, the PCA models can be seen as interface representations of each component w.r.t. to a given architectural configuration. If the Server was a composite component, its interface representation would preserve the encapsulation property of its architectural model, as the behaviour associated with interactions with its sub-components would be hidden from external components, *e.g.* Client.

After translating the composite model to a DTMC, we compute the probability of the Client-Server system failing after one request: 1%. Note that since the Client component handles the visible failure action $sendContent$, from the Server component, only the internal failure $processError$ affects the reliability properties of the Client-Server system. On the other hand, the reliability of the Client-Server system would be lower if the Client component used the interface $getImage$ of the Server component due to the higher probability of internal failure action $requestError$.

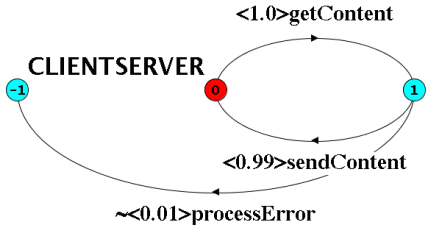

Fig. 4. Composite PCA model of Client-Server system,

We have shown how PCA can be used to model the probabilistic behaviour of component-based systems. A close link with the architectural model is needed in order to automatically construct a correct composite model for a given architectural configuration.

V. Conclusion and Future Work

Our Probabilistic Component Automata provide an expressive formalism to model the probabilistic behaviour of software 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.

Autonomous architectural assembly driven by non-functional properties requires an integration between architectural, behavioural and adaptation aspects. Compositionality of models, *i.e.* deriving a composite model from the models of its parts, is a key requirement for systems to autonomously adapt. PCA models are compositional and establish a close correspondence between behavioural and architectural aspects. Input and output actions correspond to provided and required interfaces of component models such as Darwin [10]; the architectural structure of composite components, which hides internal bindings between sub-components, is preserved at the behaviour level by applying the hiding operator to compute their interface behaviour. Therefore, the reliability of alternative architectural configurations can be automatically analysed at runtime using the corresponding composite PCA model.

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 constructed 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.

We intend to extend PCA with variables to support late specification of transition probabilities. This 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.* [25] for analysis with parametric DTMC models). These would provide means for scalable, accurate probabilistic analysis at runtime.

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] 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.

[6] 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, 2010, pp. 17–24.

[7] E. Pavese, V. Braberman, and S. Uchitel, "Probabilistic environments in the quantitative analysis of (non-probabilistic) behaviour models."

[8] 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.

[9] 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.

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

[11] 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, 2010, pp. 471–472.

[12] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Form. Asp. Comput.*, vol. 24, no. 2, pp. 163–186, Mar. 2012.

[13] F. Zhang, X. Zhou, J. Chen, and Y. Dong, "A novel model for component-based software reliability analysis," in *Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium*, ser. HASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 303–309.

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

[15] 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.

[16] 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.

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

[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, 1987, pp. 137–151.

[19] 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.

[20] 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.

[21] 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.

[22] 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, 2001, pp. 109–120.

[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, 2014, pp. 548–551.

[24] P. Rodrigues, E. Lupu, and J. Kramer, "Compositional probabilistic reachability analysis for probabilistic component automata," in *Technical Report 2014/9 - Department of Computing*. Imperial College London, 2014.

[25] 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, 2011, pp. 341–350.