On Re-Assembling Self-Managed Components

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

***Abstract*—Self-managed systems need to adapt to changes in requirements and in operational conditions. New components or services may become available, others may become unreliable or fail. Non-functional aspects, such as reliability or other quality-of-service parameters usually drive the selection of new architectural configurations. However, in existing approaches, the link between non-functional aspects and software models is established through manual annotations that require human intervention on each re-configuration and adaptation is enacted through fixed rules that require anticipation of all possible changes. We propose here a methodology to automatically re-assemble services and component-based applications to preserve their reliability. To achieve this we define architectural and behavioural models that are composable, account for non-functional aspects and correspond closely to the implementation. Our approach enables autonomous components to locally adapt and control their internal configuration whilst exposing interface models to upstream components.**

I. Introduction

Self-managed systems need to adapt their configuration at run-time to changes in requirements or in the components from which they are assembled. This is equally true in pervasive embedded systems or service-based distributed architectures where new devices or services may become available and existing ones may degrade or fail.

Current approaches to adaptation and re-assembly rely on pre-defined event-condition-action rules [1], [2], extensions of design models with simplified representations [3]–[6] or non-composable models of non-functional properties [7], [8]. They require either anticipating all possible reconfigurations or manual input from system designers to re-model the non-functional properties of the system. How can we then move towards automation?

An essential aspect is the *compositionality* of the models i.e., the ability to derive the representation of composite components from the representations of their parts. This must apply to the components' architectural, behavioural and management aspects and to the representation of their non-functional aspects. Only then can we calculate the aggregated properties of configurations in order to select between alternative ones [4], [6], [9]. Non-functional properties are often probabilistic and composing probabilistic models is difficult. Although performance models such as PEPA [10] are compositional, they impose a semantics based on the duration of actions and have limited representations for failures, their propagation and handling. In addition, they allow answering 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 the *reliability* of the system interpreted as the reachability of a failure states, represent failures, their propagation and handling in a natural way, and answer questions such as "what is the probability that the system fails?" or "what is the probability of failure after action a?". For this purpose we have defined the Probabilistic Component Automata (PCA) [11], which support the automatic construction of composite representations and advanced failure representation. Both PEPA and PCA may suffer from state-explosion when composing models, which hinders their ability for analysing large systems. We therefore defined an algorithm based on Compositional Reachability Analysis [12] to help mitigate the state explosion by reducing the size of composite PCA models.

In this paper we show how PCAs can be used effectively for distributed assembly of autonomous components. We propose a distributed assembly algorithm to automatically select re-configurations that maximise the system's reliability. We compare a centralised and a distributed version of the algorithm using an example application scenario. We further show how this algorithm can be applied as part of a more general methodology that encompasses model extraction and analysis and that integrates architectural, behavioural and management views of application components. Our methodology is applicable to component-based systems and services alike. We only assume that *required* and *provided* interfaces can be identified and probabilities of transitions can be extracted from monitoring and profiling logs. These constitute the main contributions of this paper and are presented in Sections III and IV. We also review related approaches from Software Engineering and Systems Management in Section II and evaluate the performance of the assembly algorithms in Section V. Conclusions are presented in Section VI.

II. Related Work

Architectural adaptation consists in changing the bindings between components or replacing the components (services) themselves. Several approaches associate non-functional parameters with components and select configuration changes based on aggregated measures of the non-functional parameters of their components. Sykes *et al.* [9] consider the components independently, Grassi *et al.* [4]; extend theis to account for dependencies between components. But neither consider the behavioural aspects, which are necessary to assess the system's reliability. Indeed, a component's reliability depends on how and which parts of its behaviour are used. Feature driven models [5], [6] consider alternative implementations that can be changed at run-time. But they are not composable and cannot be easily analysed.

Across most existing approaches non-functional properties are, in essence, manually defined annotations. Although in [7] DTMC models of the system are updated at runtime, this approach treats components as black-boxes and the DTMC model of a composite component cannot be automatically constructed from models of its sub-components. Thus, a new DTMC has to be manually defined for each architectural configuration. In contrast, PCA [11] supports automatic construction of composite representations and this enables analysing and selecting the most appropriate configuration automatically. Showing how this is achieved is one of the aims of this paper.

The integration of previous approaches with management aspects is also limited. Self-management frameworks implement variations of a MAPE loop (Monitoring, Analysis, Planning and Execution). The Self-Managed Cell (SMCs) [1] supports distributed and composable autonomous components through Event-Condition-Action (ECA) policies. Rainbow [2] integrates architectural models with ECA policies that change system parameters on the basis of their costs and benefits given as annotations. But in such systems changes must be anticipated in advance when policies are defined.

We build here upon our PCA formalism to show how component re-assembly can be self-managed and automated based on both architectural and behavioural models. PCA models can, to a large extent, be extracted from the software implementation. Their transitions are probabilistic and can be derived from system execution and profiling logs. They can represent failure handling and failure propagation to express reliability concerns. PCA models are composable and can be reduced to a component's interface behaviour. This allows us to analyse models to compare architectural configurations and select the most appropriate one. We show hereafter how re-assembly can be achieved in a distributed fashion where each component autonomously decides on its internal configuration.

III. Notational Elements for Distributed Architectural Adaptation

Component-based models can be reconfigured by changing the bindings between *provided* and *required* interfaces. We use *component* in a generic sense to denote encapsulation of behaviour. Components could equally well be libraries, hardware devices or (web) services. We assume that components are composable *i.e.* a composite component is realised as a configuration of sub-components. When multiple configurations achieve the desired functional behaviour, non-functional properties such as reliability or other quality-of-service parameters are used to select the configuration to deploy. Management services monitor system behaviour, measure non-functional parameters and enact reconfiguration changes to preserve the system's quality of service or reliability. Component-based applications and services can therefore be seen from an architectural, behavioural or management perspective.

A. Architectural View

The architectural view comprises the components with their *provided* and *required* interfaces and their bindings; for composite components, sub-components and internal bindings are also specified. Components provide encapsulation of behaviour and autonomy: each component can be seen as managing its internal configuration of sub-components and adapting it to achieve the required reliability. Similarly to [9], we use a notation based on Darwin [20] to graphically represent the architecture of the application. Other notations exist and are broadly equivalent. However, in contrast to [9] we derive non-functional parameters such as the reliability of a composite component are automatically derived from the models of its sub-components.

B. Behavioural view

The behavioural view comprises the representation of the components' probabilistic behaviour described as PCAs [11]. We distinguish between *generative* transitions which model internal actions or output actions and *reactive* transitions that correspond to input actions exposed through a provided interface. Tool support for PCA representation and analysis is presented in [21]. Formally, a PCA 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 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.

PCA models correspond therefore closely to the architectural models. They also correspond closely to the source code, thus producing a faithful representation of the implementation. In essence, an output action corresponds to calling an external method, whereas input actions correspond to methods of a component's interface being called.

We introduce failure actions to model failure scenarios, failure propagation and failure handling in a manner that is analogous to the conventional use of exceptions in object-oriented programming languages. If a PCA is in state s and can execute an unreliable internal action e, a failure 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, *output failure actions* $(s, \sim!e, \texttt{ERROR})$ model externally visible failures such as communication failures, and *input failure actions* correspond to the handling of an output failure; details are given in [11].

The behaviour representation of composite components is automatically constructed as the parallel composition of the models of its sub-components. Input and output actions are synchronised to model interactions between two components, and internal actions are interleaved to model concurrent execution.

In the architectural view, the interfaces of a component hide its internal behaviour. Similarly, when applied to a PCA A, the hiding operator $\backslash\{a_1, \ldots, a_n\}$ collapses, when possible, transitions in A labelled with actions $\{a_1, \ldots, a_n\}$, while maintaining the probabilistic reachability properties of the original process. *Hiding* can be used to remove behaviour associated with unbound provided interfaces, internal transitions or reduce a PCA to its interface behaviour. Its dual, the interface operator $@\ \{a_1, \ldots, a_n\}$, indicates the transitions that should be kept and is equivalent to $\backslash\Big\{\mathcal{E}_A - \{a_1, \ldots, a_n\}\Big\}$. We have defined a probabilistic extension of the CRA algorithm to implement the hiding and interface operators; for further details see [11].

We have implemented PCAs in the LTSA tool which we have extended to automatically construct composite representations from the PCA models of each component [21] . This allows us to construct and analyse models for composite components and entire systems. In the latter case, when the composite PCA is *closed* i.e., it does not have any unbound input interfaces, the PCA model can be translated to a DTMC model which we then analyse with the PRISM model checker [22] to determine its reliability. Sensitivity analysis on the translated model can determine the impact of changes on: *a)* the probability of failure of a component; *b)* clients execution profile and *c)* bindings configuration.

C. Management View

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

Components can be self-managed *i.e.* implementing their own management services and managing their own internal configuration of sub-components, we refer to reconfiguration in this case as *distributed assembly* (section IV), or can be managed by an external management system which determines the reconfiguration of a whole component hierarchy in a *centralised* case (section IV-A).

1) Monitoring: Monitoring the execution of a component is necessary to update the transitions probabilities μ in the behavioural models of all transitions including failures.

2) Instance Inventory: Re-configuring a system requires knowing at all times which components are available and thus maintaining an inventory of available components and their characteristics. Typically, this requires discovering new components when they become available and detecting their failure or absence.

3) Decision-Making: When re-configuration is needed to preserve reliability requirements, a decision-making function is necessary to identify the changes to make. In our case, this entails selecting re-configurations that are most reliable and meet reliability requirements.

IV. Non-functional Architectural Assembly

Automating architectural assembly requires determining the possible alternative configurations based on the available components, calculating their reliability properties and choosing the most reliable configuration that satisfies requirements. Calculating the reliability properties of a configuration requires in turn: reducing the PCA model of each component to its interface representation using the hiding operator and then constructing the composite model from the reduced models of the components.

In the following, we contrast centralised assembly with distributed assembly. In the latter, each component behaves autonomously, computes its internal most reliable configuration and exchanges its interface models with other components.

A. Centralised Assembly

Consider a set of available components $\mathcal{C} = \{C_1, \ldots, C_n\}$, where each C_i is associated with a set of provided interfaces $\mathcal{P}_{C_i} = \{p_1, ..., p_n\}$ and a set of required interfaces $\mathcal{R}_{C_i} = \{r_1, ..., r_n\}$. An architectural configuration is then defined by the set of components $\mathcal{C}_{arch}$ and the bindings between their provided and required interfaces $\mathcal{B}_{arch} = \{C_i.p_j - -C_k.r_j\}$, where $C_i.p_j$ ($C_k.r_j$) denotes the provided (required) interface p_j (r_j) of component C_i (C_k). The probabilistic behaviour of component C_i is then defined by the PCA: $\mathcal{A}_{C_i} = \langle S_{C_i}, q_{C_i}, \mathcal{E}_{C_i}, \Delta_{C_i}, \mu_{C_i}\rangle$.

The possible architectural configurations can be obtained from the available component instances $\mathcal{C}$ by using a constraint solver [9]. For each configuration $arch$, if the provided interfaces of the components in $\mathcal{C}_{arch}$ are all bound to required interfaces of components in $\mathcal{C}_{arch}$, then the composite behavioural representation for $arch$ is given by the parallel composition of the PCA models of its components: $\mathcal{A}_{C_1} \| \ldots \| \mathcal{A}_{C_n}$, $\mathcal{A}_{C_i} \in \mathcal{C}_{arch}$, $i \in \{1, \ldots, n\}$, $n = \#(\mathcal{C}_{arch})$.

However, in general, not all provided interfaces may be bound as some functionality may not be used. We therefore reduce $\mathcal{A}_{C_i}$ to an interface model based upon the bound interfaces of C_i using the hiding operator. If $\mathcal{B}(C)$ denotes the interfaces of component C used in configuration $\mathcal{B}$, and $\mathcal{E}(C.i)$ the set of actions of interface i in component C, then the *interface process* of C for configuration $\mathcal{B}$ is given by:

$$\mathcal{I}_C = \mathcal{A}_C \setminus \Big(\mathcal{E}_C - \bigcup_{i \in \mathcal{B}(C)} \mathcal{E}(C.i)\Big).$$

Thus, the hiding operator is used to produce for each component C in $arch$ a representation consistent with the bindings in $\mathcal{B}_{arch}$. This reduces significantly the size of the composite model as it removes all interleaving of internal and unbound transitions; we have obtained reductions in size of around 90% that translate in reduced complexity and shorter analysis time. If $arch$ does not have unbound provided interfaces, the composite PCA can be translated into a DTMC for reliability analysis based upon the reachability of failure states [11].

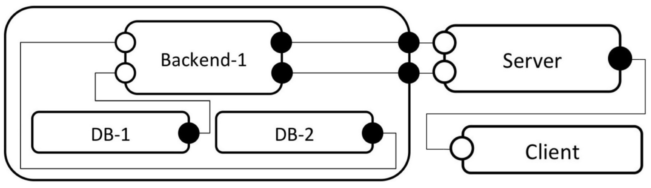

Fig. 1. Example Web system

Consider a simple Web system (Figure 1) consisting of a $Client$ that requests pages from a $Server$, which uses a backend infrastructure to obtain dynamic text content and images. The backend is implemented as a composite component that relies on two databases (DB-1 and DB-2). This system is instantiated with the following bindings configuration $\mathcal{B}_1$:

$$\begin{array}{lcl}
Server.getWebPage & -- & Client.getWebPage \\
Backend\text{-}1.getImage & -- & Server.getImage \\
Backend\text{-}1.getContent & -- & Server.getContent \\
Backend\text{-}1.getContentDB & -- & DB1.getContentDB \\
Backend\text{-}1.getImageDB & -- & DB2.getImageDB
\end{array}$$

Since there are no unbound interfaces, its composite PCA representation can be obtained as the parallel composition of $\mathcal{A}_{Client}, \mathcal{A}_{Server}$ and $\mathcal{A}_{Backend\text{-}1}$. When a new component $Backend$-2 that provides images from a separate database DB-3 becomes available, the following alternative configuration can be considered $\mathcal{B}_2$ (Figure 2):

$$\begin{array}{lcl}
Server.getWebPage & -- & Client.getWebPage \\
Backend\text{-}2.getImage & -- & Server.getImage \\
Backend\text{-}2.getImageDB & -- & DB3.getImageDB \\
Backend\text{-}1.getContent & -- & Server.getContent \\
Backend\text{-}1.getContentDB & -- & DB1.getContentDB
\end{array}$$

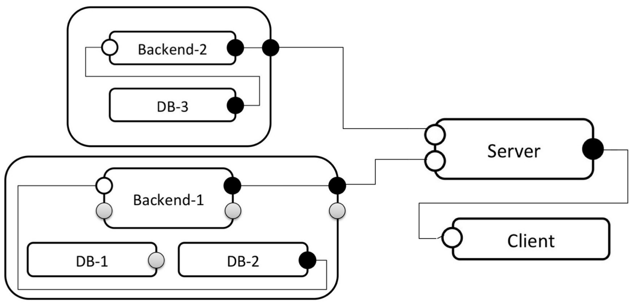

Fig. 2. Example Web system - alternative configuration

Both $Backend$-1 and $Backend$-2 provide the $getImage$ interface but the $Server$'s $getImage$ required interface is only bound to $Backend$-2. So before constructing the composite representation for $\mathcal{B}_2$ through the parallel composition of $\mathcal{A}_{Client}, \mathcal{A}_{Server}, \mathcal{A}_{Backend\text{-}1}$ and $\mathcal{A}_{Backend\text{-}2}$ the PCA representation of $Backend$-1 needs to be reduced to include only the interfaces used in $\mathcal{B}_2$:
$\mathcal{I}_{Backend\text{-}1} = \mathcal{A}_{Backend\text{-}1} \setminus \big(\mathcal{E}(getImage) \cup \mathcal{E}^{int}_{Backend\text{-}1}\big)$.

The reliability of $\mathcal{B}_1$ and $\mathcal{B}_2$ can now be calculated based on the reachability of failure states in the composite representation and the best configuration can be selected.

B. Distributed Assembly

Centralised assembly assumes that a central point knows all components, their architecture, configuration and behaviour. However, in many cases components need to make autonomously local re-configuration decisions. This requires them to exchange information regarding their interfaces and behaviour so that each component can analyse its resulting reliability properties. We propose here an algorithm for distributed re-assembly (Algorithm 1), which is discussed later in this section. We start however by illustrating it in the case of the Web system example.

The initial components $Client$, $Server$ and $Backend$-1, have a single possible configuration given in $\mathcal{B}_1$. Therefore, in the initial run of the algorithm, each component sends to the components that can be bound to its required interfaces a reduced PCA model, which describes how it uses their

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

Algorithm 1: Distributed Assembly Algorithm

provided interfaces. When $Backend$-2 becomes available, the $Server$ can consider the following alternative bindings to its required interfaces (Algorithm 1 - line 11):

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

The reliability of each configuration is then analysed by the $Server$ based on how its provided service is used by the $Client$ (algorithm 1 - line 3). The behavioural representation for $\mathcal{B}_1$ has been calculated in the first run so we describe the steps to calculate that for $\mathcal{B}_2$ referring to the relevant lines in Algorithm 1. The $Server$ first uses the hiding operator to compute its PCA representation for each component bound to it in $\mathcal{B}_2$ (line 15) *i.e.* to Backend-1 and Backend-2 and sends it to them (line 16). For instance, the representation for $Backend$-1 denotes how the $Server$ uses the $getContent$ interface.

$Backend$-1 receives the reduced PCA from $Server$ (line 1), and uses it to construct a PCA representation of its behaviour as used by the server and its required interfaces (line 2). It applies the hiding operator to remove the behaviour of its unbound provided interfaces (*i.e.* methods that are not called such as $getImage$). It then identifies which of its required interfaces need to be bound (lines 4-7) and the components that can provide them. In our example it requires DB-1 to provide $getContentDB$ but not DB-2 to provide $getImageDB$. $Backend$-1 then sends DB-1 a reduced PCA representation which denotes how it uses the interface $getContent$, given how itself is used by the $Server$ (lines 15-16). DB-1 is a leaf component, *i.e.* it does not have required interfaces, so replies sending its interface PCA representation for $Backend$-1 (lines 27 and 28). This allows $Backend$-1 to compute its behaviour representation given its downstream dependencies (DB-1) reduce it to its interface representation w.r.t to the $Server$ (line 26) and send it to the $Server$. $Backend$-2 performs similar steps to provide its behaviour (given its downstream dependences) to the $Server$.

Having received the representations from $Backend$-1 and $Backend$-2 (line 17) for the configuration $\mathcal{B}_2$, the $Server$ can combine them with its own behaviour to compute the composite representation for $\mathcal{B}_2$ (line 18). This representation is *closed i.e.* all interfaces are bound, can therefore be translated to a DTMC to calculate the resulting system reliability properties for each of the configurations (line 20). The Server is then ready to compare the reliability of the configurations $\mathcal{B}_2$ and $\mathcal{B}_1$, which had been previously calculated. Since $\mathcal{B}_2$ achieves a higher reliability than $\mathcal{B}_1$, it is chosen by the Server who then calculates a composite representation w.r.t. to the interfaces used by $Client$ (lines 26 and 27) and sends it to the $Client$.

C. General algorithm description

In the general case, the distributed assembly algorithm starts a descending phase from a component C_1 with no provided interfaces, *i.e.* the root of a composition hierarchy. C_1 calculates all its possible bindings given its required interfaces and available sub-components and sends to each sub-component and for each configuration, its the behaviour model dictating how it would use their provided interfaces. This descending phase is repeated and stops when *leaf* components are reached. An ascendent phase is then started where each component selects the configuration that maximises its reliability, and propagates that choice upwards in the hierarchy.

These computations do not need to be repeated at each configuration change. For example, if a component C becomes unavailable, the component(s) that used C have to select an alternative configuration to meet their functional requirements. As these components instances have calculated all the possible configurations w.r.t. to their required interfaces (lines 11-22), no computation is required for downstream dependencies. However, as the new configuration is less reliable, these component instances need to send their new reduced PCA to their upstream dependencies (lines 24-26), which may trigger changes in the configuration of upstream composite components. Similarly, when a new component C becomes available, existing component instances that can use its functionality can select a configuration with better reliability and send it to their upstream dependencies.

When the execution profile of a component C changes, the transition probabilities in its PCA model and consequently the reliability of the configurations involving C. To calculate the new reliability values, C constructs a new reduced PCA (line 2) and propagates the new representation first to downstream dependencies (lines 3 - 23), and then to the upstream components that use its provided interfaces.

Although the same PCA model of a component is used in all configurations and restricted using the hiding operator, the reliability provided by a component differs in each configuration as it is determined by how the component is used by upstream components and the reliability provided by its downstream dependencies. For example, a component may fail on a specific action with a high probability but that action may be rarely used in a given configuration so will not have a significant impact. While the probability of a transition is given by the component's execution profile, the frequency with which it is called also depends on the probability with which relevant component interfaces are called. Therefore, a component's reliability is always w.r.t. a particular configuration. The descending phase of the algorithm ensures that each component knows how it is going to be used by upstream components and the hiding operator ensures that only interface actions are exchanged between components. In the ascending phase, the composite representation computed contains the propagation and handling of failures from downstream dependencies. So the reliability of downstream dependents is combined with the execution profile of upstream components propagated during the descending phase and each component has all the information required.

V. Evaluation

Of particular interest is the overhead introduced by the *distributed* re-assembly algorithm since it computes intermediate representations at each descending and ascending step. We use the *centralised* algorithm as a baseline for comparison.

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

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

Although this is only small example and further evaluation at larger scales is certainly needed, the figures suggest that the overheads of computing intermediate representations are not prohibitive and can be easily compensated if the components execute concurrently. This is due to the fact that each

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

TABLE I. ARCHITECTURAL ASSEMBLY EVALUATION RESULTS

component's behaviour has been reduced to its interface representation which remains small. Indeed, constructing reduced interface models offers significant gains over the use of full behavioural models that suffer from state explosion [11].

Another important aspect of reducing the behaviour to interface models is that it allows the suppliers of different autonomous components such as hardware devices to exchange models that do not expose their underlying (and often proprietary implementations), which cannot be removed from the PCA representation of such components as it is necessary to monitor the execution profile of internal computations.

VI. CONCLUSIONS

(Self-) adaptation of autonomous systems, whether distributed applications or services is driven by non-functional aspects that need to be automatically (rather than manually) associated with their software models. This requires an integration between the architectural, behavioural and management concerns. Models that ignore one of them - for example information models tend to ignore behavioural aspects - must rely on user input and pre-defined scenarios of change. We ensure, a close correspondence between the elements of the different views: bindings and synchronised actions, provided (required) actions and reactive (generative) transitions, execution monitoring and transition probabilities.

Compositionality of models – deriving a composite model from the models of its parts – is a key requirement for autonomous systems to be able to reason and adapt. However, non-functional aspects are often probabilistic and composing probabilistic models can be challenging. We have introduced PCA to address some of the shortcomings of existing models and to be able to represent failures, their propagation and handling.

Self-management, requires encapsulation of the components' internal behaviour. This is needed for commercial as well as technical reasons: internal behaviour should not be disclosed. But reducing probabilistic models to their interface behaviour can be challenging. For example, when deleting a probabilistic transition where should its probability be propagated? Our algorithm for reducing probabilistic models forms an important part of our solution.

When architectural, behavioural and management models are integrated, when models are composable and can be reduced to their interfaces, it is possible for systems made of autonomous components to re-configure themselves to preserve global non-functional requirements. We have focused in this paper on reliability analysis based on reachability of failure states but other non-functional properties can be similarly represented.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] M. Sloman and E. C. Lupu, "Engineering policy-based ubiquitous systems," *Comput. J.*, vol. 53, no. 7, pp. 1113–1127, 2010.

[2] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, pp. 46–54, October 2004.

[3] D. Sykes, J. Magee, and J. Kramer, "Flashmob: Distributed adaptive self-assembly," in *Proc. 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '11, 2011, pp. 100–109.

[4] V. Grassi, M. Marzolla, and R. Mirandola, "Qos-aware fully decentralized service assembly," in *Proc. 8th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).*

[5] A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: a framework for engineering self-tuning self-adaptive software systems," in *Proc. ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE).*

[6] M. Mori, F. Li, C. Dorn, P. Inverardi, and S. Dustdar, "Leveraging state-based user preferences in context-aware reconfigurations for self-adaptive systems," in *Proc. 9th Int. Conf. on Software engineering and formal methods*, ser. SEFM'11, 2011, pp. 286–301.

[7] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *ICSE*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–121.

[8] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.

[9] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "Exploiting non-functional preferences in architectural adaptation for self-managed systems," in *Proc. of the 2010 ACM Symp. on Applied Computing*, ser. SAC '10, 2010, pp. 431–438.

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

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

[12] 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.

[13] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The ponder policy specification language," in *Proc. Int. Workshop on Policies for Distributed Systems and Networks*, ser. POLICY '01, 2001, pp. 18–38.

[14] G. Tesauro and J. O. Kephart, "Utility functions in autonomic systems," in *Proc. 1st Int. Conf. on Autonomic Computing*, 2004, pp. 70–77.

[15] A. J. Ramirez, B. H. Cheng, P. K. McKinley, and B. E. Beckmann, "Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration," in *Proc. 7th Int. Conf. on Autonomic computing*, ser. ICAC '10, 2010, pp. 225–234.

[16] M. Derakhshanmanesh, M. Amoui, G. O'Grady, J. Ebert, and L. Tahvildari, "Graf: graph-based runtime adaptation framework," in *Proc. of the 6th Int. Symp. on Software engineering for adaptive and self-managing systems*, ser. SEAMS '11, 2011, pp. 128–137.

[17] C. Ghezzi, J. Greenyer, and V. Manna, "Synthesizing dynamically updating controllers from changes in scenario-based specifications," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, june 2012, pp. 145 –154.

[18] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio, "Reliability-driven dynamic binding via feedback control," in *7th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, Jun. 2012.

[19] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke, "Viewpoints: A framework for integrating multiple perspectives in system development," *Int. J. of SW Eng. and Knowledge Engineering*, vol. 2, no. 1, pp. 31–57, 1992.

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

[21] P. Rodrigues, E. Lupu, and J. Kramer, “Ltsa-pca: Tool support for compositional reliability analysis,” in *Companion Proceedings 36th Int. Conf. on Software Engineering (ICSE)*.

[22] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *CAV’11*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, pp. 585–591.

[23] L. de Alfaro and T. A. Henzinger, “Interface automata,” in *Proc. 8th Eur. SW Eng. Conference (ESEC/FSE), isbn = 1-58113-390-1, location = Vienna, Austria, pages = 109–120, numpages = 12, 10.1145/503209.503226, New York, NY, USA,*.

[24] E. W. Stark and G. Pemmasani, “Implementation of a compositional performance analysis algorithm for probabilistic i/o automata,” in *In Proc. Work. on Process Algebra and Performance Modelling (PAPM)*, 1999, pp. 3–24.

[25] I. Krka, L. Golubchik, and N. Medvidovic, “Probabilistic automata for architecture-based reliability assessment,” in *Proc. 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (QUOVADIS*.

[26] 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.