Oliveira and Barbosa *Journal of Software Engineering Research and Development* (2015) 3:6
DOI 10.1186/s40411-015-0021-2

Journal of Software Engineering Research and Development
a SpringerOpen Journal

RESEARCH Open Access

Self-adaptation by coordination-targeted reconfigurations

Nuno Oliveira* and Luís S Barbosa

*Correspondence:
nuno.s.oliveira@inesctec.pt
HASLab - INESC TEC & Universidade do Minho, Braga, Portugal

Abstract

Background: A software system is *self-adaptive* when it is able to dynamically and autonomously respond to changes detected either in its internal components or in its deployment environment. This response is expected to ensure the continuous availability of the system by maintaining its functional and non-functional requirements.

Methods: Since these systems are usually distributed, coordination middleware (typically a centralised architectural entity) plays a definitive role in establishing the system goals. For these reasons, adaptations may be triggered at coordination level, issuing reconfigurations to such a coordination entity. However, predicting when exactly reconfigurations are needed, and if they will lead the system into a non disruptive configuration, is still an issue at this level. This paper builds on a framework for formal verification of architectural requirements, either from a qualitative or quantitative (probabilistic) point of view, which will leverage analysis and adaptation prediction.

Results: In order to address the mentioned difficulties, it is discussed both a model that lays down reconfiguration strategies, planned at design time, and a process that actively uses such a model to trigger coordination-targeted reconfigurations at run time. Moreover, a cloud-based architecture for the implementation of this strategy is proposed, as an attempt to deliver adaptation as a service. A case study is presented that assesses the suitability of the approach for real-world software systems.

Conclusions: We highlight the use of formal models to represent the coordination layer and necessary reconfigurations of a software system, and also to predict the need for (and to trigger) adaptations.

Keywords: Self-adaptive software; Feedback loop; Reconfiguration; Software coordination; Service-oriented architectures

1 Introduction

Emergency call-centers facing unexpected peaks of activity, surveillance systems whose CCTV devices have to operate under changeable environment conditions, or applications for mobile devices constrained by limited battery autonomy, are examples of systems which have somehow to adapt to change along a normal operating cycle. The expression *self-adaptive* qualifies a behaviour which has to respond at run time to contextual changes, detected either internally or externally, in order to keep meeting its own functional requirements and general service level agreement (SLA), ensuring the relevant quality of service (QoS) attributes (Garlan et al. 2009; Oreizy et al. 1999).

This entails the need for some degree of introspection. Actually, such systems should be able to keep track of their internal interconnection structures, attributes, execution environment, requirements and reference performance levels; but above all, to observe and detect changes in these elements. Such observations, suitably processed (*e.g.*, by comparison to reference levels assigned to measurable variables) will be responsible for triggering adaptations.

This process, which spans from acquiring information to check for relevant changes, to actually enacting adaptations, is known as the *control* or *feedback loop* model in the literature associated to control theory, autonomic computing, robotics or artificial intelligence (Gat 1998; Nilsson 1980). Its implementation involves four components responsible for monitoring, analysing, planning and executing changes, as defined in the MAPE(-K) reference model (IBM Corp 2004; Kephart and Chess 2003). In self-adaptive software this model is realised by monitoring the environment and probing the system's attributes; analysing the data collected to infer situations in need for adaptation; deciding the adaptation strategy; and finally, enacting reconfigurations to enforce the system's adaptation into acceptable (non disruptive) configurations (Brun et al. 2009; Dobson et al. 2006; Villegas Machado et al. 2011).

Self-adaptive systems are often distributed, component-based, with highly demanding requirements. Coordination middleware, typically a centralised architectural entity, defines the interaction between such components. This is responsible for establishing the overall system goals by covering its requirements (Arbab 2004). For this reason, the coordination layer of these systems plays a fundamental role in the adaptation process. Concretely, coordination models (*e.g.*, Reo (Arbab 2004), BIP (Basu et al. 2011), among others) are operative in the generation of introspective/reflective abstractions of the whole system from its coordination layer. This highlights the importance of coordination-targeted reconfigurations.

But deciding and applying reconfigurations is not an easy task. Mainly, this is due to the unpredictable, evolutive nature of the deployment context, which precludes knowing with exactitude when a reconfiguration has to be applied, and predicting its outcome. Reconfigurations can be planned in advance provided that a number of relevant context attributes are identified and translated into measurable variables. Suitable ranges of values for these attributes may help to plan (at design time) configurations that will, most likely, drive the system into stable states meeting specific sets of conditions. Nevertheless, assumptions made at design time may not apply directly after deployment. On the other hand, unpredictable contexts may trigger reconfigurations that were not intended to occur because, for example, they may violate some key functional properties of the original design. Triggering reconfigurations must, therefore, take into account, not only the expected QoS levels, but also functional properties which are identified as design invariants.

We have recently proposed a framework for modelling coordination-targeted reconfigurations and verifying their properties in the presence of contextual changes (Oliveira and Barbosa 2013a, b). This work is based on a generic coordination model encompassing a graph whose edges are regaded as connectors specifications. The properties of interest are relative to *behaviour*, classifying reconfigurations *w.r.t* behavioural changes provoked to the coordination model; or to *structure*, namely to the topology of the underlying coordination model. Structural properties are expressed in a specific variant of hybrid logic

(Blackburn 2000), interpreted over the graph representing the interconnection network. We have also introduced a quantitative behavioural model (Oliveira et al. 2014) for this coordination style based on Markov chains (Hermanns 2002). This opened the possibility to assess and compare reconfigurations along a quantitative (actually, a probabilistic) reasoning dimension.

In broad terms, this paper focuses on the *adaptability* quality attribute for software architectures, often regarded as a major one in architectural design (Ciraci and van den Broek 2006; Losavio et al. 2003). In particular, the paper proposes a self-adaptation strategy, following the MAPE reference model. The novelty is the introduction of a model of coordination-targeted reconfiguration strategies, planned at design time. This model is actively used to decide and trigger adaptations at run time. The model's key ingredient is a transition system whose states are the configurations originally envisaged for the architecture, and edges represent reconfigurations, *i.e.*, paths from a configuration to another.

The work reported here extends the original SBCARS'2014 paper (Oliveira and Barbosa 2014) as follows:

- a state transfer strategy for dynamic reconfigurations is formalised,
- the self-adaptation strategy is detailed,
- an architecture to deliver adaptation as a cloud-based service is proposed.

The envisaged adaptation strategy is discussed in Section 4. Before that, in Section 2, the underlying framework for reconfigurations is introduced; and in Section 3 this is further extended to cope with dynamic reconfigurations, notably with the consistent state transfer problem. A detailed example is discussed in Section 5. Section 6 proposes a refactoring of the adaptation strategy in order to deliver it as a cloud-based service for adaptation. Section 7 revises relevant related work; and finally, Section 8 concludes and proposes topics for future work.

2 A framework for architectural reconfiguration

A software architecture is often represented as a graph whose vertices are labelled by components and interconnected by adapters, wrappers, connectors or other forms of *glueware* depicted in the edges (Wermelinger 1999). In this setting, architectural reconfigurations mainly target components and connectors by adding, removing or substituting them as blocks (Hnětynka and Plášil 2006). However, in typical service-oriented systems, the coordination layer becomes prominent. Therefore, our focus will be the reconfiguration of connectors and the communication protocols they implement.

2.1 Modelling

As proposed in (Oliveira and Barbosa 2013a, b), software architectures are regarded as graphs of communication channels, where nodes are interaction points and edges are labelled with an identifier and a type which encodes a concrete coordination policy. These graphs are called *coordination patterns* and concretely model service orchestration. They are abstract representations of software connectors and therefore independent of any concrete coordination model. Each coordination pattern is characterised by its input and output ports and the internal interaction of channels, which provide them with a specific behaviour. The set of all coordination patterns is denoted by $\mathcal{P}$. Fig. 1a depicts an example

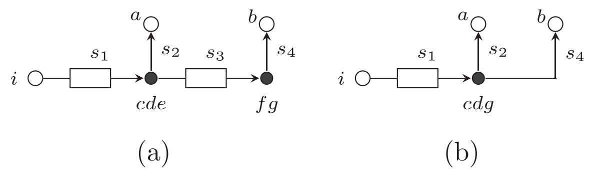


Fig. 1 Coordination patterns example. The Sequencer (**a**) and the ParallelSplit (**b**). White circles denote interface ports while black circles stand for internal nodes

of a coordination pattern which allows for a sequential interaction on output ports a and b after a stimulus is received on input port i. Fig. 1b, in turn, depicts a coordination pattern that ensures a parallel interaction on output ports a and b, after being stimulated on input port i. For concreteness, the Reo framework (Arbab 2004) has been adopted to type channels and to represent them graphically.

In this context, a reconfiguration is defined as any change made to the structure of a coordination pattern. Such changes are guided by the application of primitive operations that manipulate the pattern's basic elements. An algebra of reconfigurations was defined based on the following primitive reconfiguration operations: const_π, par_π, join_N, split_n and remove_c, where indexes represent parameters: π is a coordination pattern, N is a set of nodes, n is a node and c is a channel identifier. The set of primitive operations is denoted by *Prim*.

These operations are applied sequentially to a coordination pattern. An intuitive description of their behaviour is as follows: const_π substitutes π_i by π; par_π sets π in parallel with π_i (which are assumed to be completely disjoin), but does not establish any connections between the two; join_N connects all nodes in N (that exist in π_i) into a single one; split_n, as its name suggests, performs the inverse operation; and, finally, remove_c removes the channel identified by c from π_i.

These primitives may be composed sequentially to yield complex and yet reusable constructions referred to as *reconfiguration patterns*. For instance, the $\mathsf{implode}_C$ pattern, when applied to a coordination pattern π_i, removes all channels in set C from π_i (applying the remove primitive recursively over C) and then reconnects (with join) the resulting ports. Fig. 1b shows the result of applying $\mathsf{implode}_{\{s_3\}}$ to the sequencer pattern. The interaction at ports a and b becomes parallel, instead of sequential. The reader is referred to (Oliveira and Barbosa 2013a, b) for a detailed description to this algebra of reconfigurations. The set of all reconfigurations is denoted by $\mathcal{R}$.

2.2 Reasoning

Often it becomes necessary to rule out reconfigurations that lead to system states which fail to preserve some key functional requirements of a system measured either in terms of behavioural or structural properties. The ability of inspecting these properties is, therefore, mandatory when dealing with adaptable architectures. Next we discuss three perspectives on reasoning about reconfigurations: *behavioural*, *structural* and *quantitative*.

2.2.1 The behavioural perspective

In order to reason about reconfigurations from a behavioural perspective it is necessary to fix a concrete semantic model for coordination patterns. This must encompass suitable notions of observational equivalence and refinement (often encoded as bisimulation and simulation relations), which are required to compare behaviours, typically before and after reconfiguration processes.

In this framework, reconfigurations are classified as (*i*) *unobtrusive*, when the original behaviour is completely preserved; (*ii*) *expansive*, when new behaviour is added, but still preserving the original; (*iii*) *contractive*, when part of the original behaviour is removed; and (*iv*) *disruptive*, when the original behaviour or part of it is not preserved. In practice these classifications are made *w.r.t.* a specific coordination pattern and the underlying semantic model. As an example, the reconfiguration $\mathsf{implode}_{\{s_3\}}$ is *disruptive w.r.t.* the *sequencer* coordination pattern (*c.f.*, Fig. 1) and taking Reo automata (Bonsangue et al. 2012) as a concrete semantic model.

2.2.2 The structural perspective

For structural reasoning, on the other hand, the model is the (underlying graph of the) coordination pattern itself. This is taken as the (Kripke) structure (Blackburn et al. 2001) for interpretation of a propositional hybrid logic (Brauner 2010) in which structural properties are expressed. A typical example of a structural property is the requirement that a synchronous channel has to be followed by a channel with some buffering capacity. Sentences in theis hybrid logic are given by the following grammar:

$$\phi \ ::= \ i \ \mid \ \neg\phi \ \mid \ \phi_1 \wedge \phi_2 \ \mid [K]\,\phi \ \mid \ [\![K]\!]\phi \ \mid \ @_i\phi$$

where i is a nominal (a propositional symbol that is true at exactly one node of the coordination pattern); constants true, false and the boolean connectives are defined as usual; K is a set of channel types (abbreviations '$-$' and '$-t$' refer to the whole set of channel types and that set but t, respectively). Modalities $[K]\,\phi$ and $[\![K]\!]\phi$ quantify universally over the edges of the coordination pattern and express properties of the outgoing (respectively, incoming) connections from (respectively, to) the node at which the formula is evaluated. Their duals, $\langle K\rangle = \neg[K]\neg$ and $\langle\!\langle K\rangle\!\rangle = \neg[\![K]\!]\neg$, define existential quantification over the edges of the pattern. The satisfaction operator $@_i$ redirects the evaluation of a formula to the context of a node named by nominal i.

As mentioned above, this logic is able to express rather sophisticated (structural) requirements. For example, requirement "*communication through the input port is made asynchronous*" is represented by $@_i\langle\!\langle -\rangle\!\rangle\mathsf{false} \rightarrow @_i[\,\mathsf{fifo}]\,\mathsf{true}$. Here, i is a nominal referring the node i in the patterns of Fig. 1. Indeed the formula says that if the node identified by i is an input port (*i.e.*, it has no incoming connections, formally $@_i\langle\!\langle -\rangle\!\rangle\mathsf{false}$) then all outgoing channels are of type fifo, where fifo represents a buffered (asynchronous) channel.

2.2.3 The quantitative perspective

Finally, to introduce quantitative reasoning into the framework the Kripke structure derived from the underlying coordination pattern is analysed from a stochastic point of view. As a general strategy this entails the need for a stochastic model for software connectors. In (Oliveira et al. 2014) we have proposed a compositional, quantitative semantic model for Reo like connectors, based on interactive Markov chains (IMC) (Hermanns 2002; Hermanns and Katoen 2010), from which basic features (*e.g.*, compositionality and the existence of suitable notions of bisimilarity) are inherited. Stochastic coordination patterns and their reconfigurations can thus be analysed through well-known and reliable tools for stochastic processes, namely IMCA (Guck et al. 2012), CADP (Garavel et al. 2012) and PRISM (Kwiatkowska et al. 2010).

It is worth noting that a stochastic semantics can be adapted both for behavioural (regarding connectors as stochastic devices, as in (Moon et al. 2014; Oliveira et al. 2014)) and structural reasoning (regarding the coordination pattern itself as a weighted transition system). This reduces the number of model-to-model transformations, languages and tools for expressing and verifying architectural requirements, and consequently, the number of assets used in analysis. Henceforth, the set of analysable assets will be denoted by $\mathcal{A}$.

3 Ensuring consistent dynamic reconfigurations

The application of reconfigurations to the architecture of a software system at runtime is a major and non-trivial research problem. Mainly so because reconfigurations have to be transparently applied, while the exact system execution state in which a reconfiguration is required (henceforth referred to as the *interrupted state*), is hardly known *a priori*. The qualifier *transparent* above means that the system has to change its internal configuration without service disruption during and after a reconfiguration process. This entails the need for (*i*) the atomic application of reconfigurations with roll-back mechanisms triggered when the application fails; (*ii*) resuming the execution of the system in a state that is consistent (as much as possible) with the interrupted state; and (*iii*) keeping the system in line with its functional and non-functional requirements.

The framework revisited in Section 2 mitigates some of these problems. Requirement (*i*), for example, is met because primitive reconfigurations are atomic low-level operations amenable to be rolled-back, provided the existence of associated reconfiguration monitoring mechanisms. The same happens in case (*iii*) due to the methods provided for analysis (*i.e.*, verification of structural, behavioural and probabilistic properties) and comparison of reconfigurations, which can be exploited from a static perspective.

But, certainly, the framework does not support requirement (*ii*), since it does not deal explicitly with dynamic application of reconfigurations. From a static prespective, the interrupted state is either ignored or always assumed to be the initial one. After a reconfiguration, the system is again in its initial state. For the overall analysis of the system properties this approach is reasonable. Consequently, ensuring system consistency from a static perspective of reconfigurations is not a challenge. It must be taken seriously, though, when dynamism enters the equation. The unpredictable evolution of the (relevant properties of the) deployment environment may raise the need for reconfiguration at any moment in time, regardless of the overall system state.

In the sequel we propose a simple approach to consistently transfer state between configurations. This builds on an underlying automata-based semantic model of the coordination pattern, enriched with symbolic state annotations. The enacting of reconfigurations is assumed to occur when the system enters a quiescent state, as usual in practice (Kramer and Magee 1990).

3.1 A symbolic approach to state transfer

As mentioned above, reconfigurations in this framework target the coordination layer of a system, modelled through coordination patterns. These patterns exhibit behaviours in some specific semantic model, typically automata-based, defined by the software architect. However, in order to define a strategy for consistent state transfer, it is necessary that these automata are enriched with a symbolic representation of state data. In the sequel we continue considering Reo for concretely typing the edges of a coordination pattern, and we take port automata (Krause 2011) (for its simplicity) as the underlying semantic model.

Symbolic annotations are generated by the following grammar $\mathcal{S}$:

$$s ::= \varsigma \mid \neg s \mid s \wedge s$$

where ς is an atomic symbolic state. An atomic symbolic state refers to the identifier of an edge in the coordination pattern to which data is assigned. In the concrete case of Reo, we use the identifiers of the coordination pattern edges typed with a fifo channel, as this is the only stateful channel considered in Reo.

Notice that, although the notation above is borrowed from Logic, connectives $\neg$ and $\wedge$ have a specific meaning here. Thus, $\neg\varsigma$ means that the internal state ς of the pattern has no data assigned (and therefore can be omitted from the formula), and $\varsigma_1 \wedge \varsigma_2$ means that both states have data in the context of the pattern. Moreover, it is asserted that

- $\neg\varsigma_1 \wedge \varsigma_1 = \neg\varsigma_1$
- $\neg(\varsigma_1 \wedge \varsigma_2) = \neg\varsigma_1 \wedge \neg\varsigma_2$.

Additionally, notation $\perp_\pi$ is used to express that there is no data in any internal state of pattern π and $\top_\pi$ for its dual. The index π can be omitted when clear from the context.

Definition 1 (Symbolic Port Automata). *A symbolic port automaton $\mathcal{A}_\varsigma$ is an automaton $(Q, P, \rightarrow, q_0)$, where $Q \subseteq \mathcal{S}$ is a set of symbolic states, P is a set of ports, $q_0 \in Q$ is the initial state, and $\rightarrow \subseteq Q \times 2^P \times Q$ is a transition relation.*

A transition $(q, \{a, b, ...\}, p)$, written as $q \xrightarrow{\{a,b,...\}} p$, means that the system evolves from state q to state p when ports $a, b, ...$ can interact synchronously. Notation $[\![\pi]\!]_\varsigma$ is used, henceforth, to refer to the symbolic port automaton of coordination pattern π.

As an example, consider the ParallelSplit coordination pattern in Fig. 1b. The state space of the underlying symbolic port automaton is $Q = \{\perp, s_1\}$, or optionally $Q = \{\neg s_1, s_1\}$. On the other hand, the state space of the symbolic port automaton for the Sequencer coordination pattern (Fig. 1a), would be $Q = \{\perp, s_1, s_3, \top\}$ or optionally $Q = \{\neg s_1 \wedge \neg s_3, s_1 \wedge \neg s_3, \neg s_1 \wedge s_3, s_1 \wedge s_3\}$. The corresponding port automata for the Sequencer and the ParallelSplit patterns are depicted in Fig. 2 (top and bottom, respectively). On the other hand, auxiliary operation $\mathsf{IS}([\![\pi]\!]_\varsigma)$ returns the initial state of the symbolic port automaton of coordination pattern π. Whenever π is the empty coordination pattern, IS returns the general symbolic state $\perp_\pi$. The state transfer operation is defined as follows,

Definition 2 (State Transfer). *Let π be a coordination pattern, $S_r \in \mathcal{S}$ the symbolic interrupted state for reconfiguration $r = \{r_0, r_1, \ldots, r_n\}$ (where each r_i is a reconfiguration*

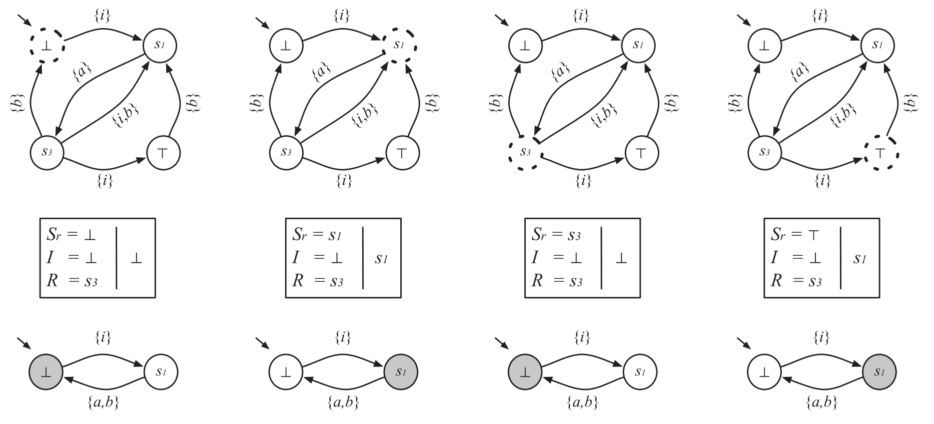

$S_r = \bot$, $I = \bot$, $R = s_3$ | $\bot$
$S_r = s_1$, $I = \bot$, $R = s_3$ | s_1
$S_r = s_3$, $I = \bot$, $R = s_3$ | $\bot$
$S_r = \top$, $I = \bot$, $R = s_3$ | s_1

Fig. 2 State transfer upon dynamic reconfiguration. Port automata for the Sequencer (top) and ParallelSplit (bottom) coordination patterns

primitive). The state transfer operation for applying r to π in state S_r, denoted by $\rightsquigarrow_{\pi,r,S_r}$, is inductively defined as $\rightsquigarrow_{\pi,r_0,S_r} \wedge \rightsquigarrow_{\pi,\{r_1,\ldots,r_n\},S_r}$, where for each $r_i \in Prim$:

$$\rightsquigarrow_{\pi,r_i,S_r} = \begin{cases} \mathsf{IS}(\llbracket \pi' \rrbracket_{\varsigma}) & \textit{if } r_i = \mathsf{const}_{\pi'} \\ S_{\mathsf{par}_{\pi'}} \wedge \mathsf{IS}(\llbracket \pi' \rrbracket_{\varsigma}) & \\ S_{\mathsf{remove}_c} \wedge \neg c & \textit{if } \mathfrak{T}^c_\pi \in \{\mathsf{fifo}\} \\ S_{r_i} & \textit{otherwise} \end{cases}$$

and $\mathfrak{T}^c_\pi$ retrieves the type of the channel c in the coordination pattern π.

This can be generalised as follows. Assume a reconfiguration r; a (possibly empty) coordination pattern π_{in} formed either by (i) all patterns introduced by par_c primitives in r or (ii) the pattern introduced by the last const_c primitive and all patterns introduced by the sequent par_c primitives in r; a coordination pattern π_{out} as the result of applying r to π; and finally $R(\pi, \pi_{out})$ as the set of *stateful* channel names removed during the reconfiguration. Then,

$$S_r \wedge \mathsf{IS}(\llbracket \pi_{\mathsf{in}} \rrbracket_{\varsigma}) \wedge \neg \bigwedge R(\pi, \pi_{out})$$

is the generalisation of $\rightsquigarrow_{\pi,r,S_r}$. The state obtained from this operation is referred to as the *resuming state.*

3.2 An application example

Consider the Sequencer coordination pattern of Fig. 1a as the model for the coordination layer of a running system. In certain situations (*e.g.*, when servers are overloaded with user requests) the system was designed to evolve into a parallelised provisioning of its services, therefore adopting a ParallelSplit configuration for its coordination layer. This involves the application of an $\mathsf{implode}_{\{s_3\}}$ reconfiguration to the original pattern.

Since the system is running, and the contexts which trigger such a reconfiguration are unpredictable, it is necessary to take the consistency of the system into consideration. This entails the need for the correct transfer of the state to the new configuration. It is assumed (for illustration purposes) that the reconfiguration process does not fail and that the obtained configuration will maintain the invariant properties of the system.

Consider the port automaton for the Sequencer coordination pattern as depicted in the first row of Fig. 2. Four replications of the automaton are presented, representing the four possible states (circled with dashes) in which the $\mathsf{implode}_{\{s_3\}}$ reconfiguration can be issued. After reconfiguration, such states must be restored if possible.

The *resuming states* in the context of the ParallelSplit port automaton are depicted as shaded circles. The tables between the automata present values for S_r, the state interrupted for application of reconfiguration $r = \mathsf{implode}_{\{s_3\}}$; I, the initial state of the structure added to the pattern; and R, the conjunction of the identifiers of stateful channels (fifo channels in this case) removed from the original pattern. These are the necessary ingredients to apply the general state transfer operation in order to obtain the desired *resuming state*.

Recall that the $\mathsf{implode}_{\{s_3\}}$ operation may be translated into the sequence of primitives $r = \left\{\mathsf{remove}_{s_3}, \mathsf{join}_{\{cd,f\}}\right\}$. Therefore, the only statefull channel removed is exactly s_3, thus $R = \bigwedge R(\pi, \pi_{out}) = s_3$, and no patterns are added to the original one, thus $I = \mathsf{IS}(\llbracket \pi_{in} \rrbracket_{\varsigma}) = \bot_{\pi_{in}}$. For the latter, since all the patterns added by $\mathsf{par}_{\{cd,f\}}$ are disjoint from the original pattern, then all symbolic states negated in $\bot_{\pi_{in}}$ are different from the ones in the original pattern.

Let us now discuss the four situations depicted in Fig. 2, from left to right, in more detail. In the first situation the reconfiguration is applied when the pattern is in its initial state. In this case such state is $\bot$, meaning that no data is assigned to the statefull channels of the pattern. Thus, the resuming state is still $\bot$ in the new configuration. In the second situation the reconfiguration is applied when the system is in state s_1. Hence, the resuming state is $s_1 \wedge \bot_{\pi_{in}} \wedge \neg s_3 = s_1$.

There are situations, though, in which it is not possible to find a suitable resuming state on the new configuration. When such is the case, the usual approach is to start the execution of the reconfigured system from its initial state. Our approach is more comprehensive on this aspect: it automatically delivers the state that best approximates the desired one. For instance, in the third situation the resuming state should be s_3. But, since s_3 is removed, the best approximated state in the port automaton of ParallelSplit is the initial $\bot = s_3 \wedge \bot_{\pi_{in}} \wedge \neg s_3$. For the same reason, in the fourth case, the interrupted state can not be resumed as is. However, in this case, the best approximated state is $s_1 = s_1 \wedge s_3 \wedge \bot_{\pi_{in}} \wedge \neg s_3$.

4 Self-adaptation strategy

The self-adaptation strategy proposed in the sequel is organised around two main phases. One is offline and concerns the planning of possible reconfigurations by the software architect. The other is online and focuses on the autonomous selection of reconfigurations to adapt a running system as part of a monitoring feedback loop.

4.1 The offline phase: planning reconfigurations

At this phase the architects have a preponderant role in preparing adaptation assets that in the online phase are autonomously used. One of these assets is a faithful model of the system architecture. This is modelled by coordination patterns (as discussed in Section 2) and constitutes the initial specification of the system. It is also in this phase that the system (functional and non-functional) requirements are encoded into verifiable properties targeting behaviour, structure and QoS. The set of all properties over the system and the environment is denoted by $\mathcal{P}rop$. In fact, this set is divided into four parts containing functional (FUN) and non-functional (QoS) properties, system generic properties (SYS) and environment specific properties (ENV). Upon these properties, the architect defines the adaptation logic as a set of constraints.

Definition 3 (Constraint). *A constraint is a triple* (ϕ, β, υ)*, where* $\phi \in \mathcal{P}rop$*;* β *is a boolean operator; and* $\upsilon \in \mathbb{R} \cup \mathbb{B}$ *is the expected value for the conjugation property-operator.*

The set of all possible constraints will be denoted by Ξ. Constraints and their utility is further addressed in Section 4.3.

The final asset from this phase is concerned with preparing (modelling and analysing) reconfigurations. The architects plan them by taking into account both the system requirements and possible ranges of values for the attributes that characterise its environment. This leads to a set of possible configurations and reconfigurations with a dependency relation between them. Such a dependency relation is captured by a *reconfiguration transition system* (RTS). Formally,

Definition 4 (RTS). *A RTS is a tuple* $(C, \rightarrow, k_i)$*, where* $C \subset \mathcal{P} \times 2^{\Xi} \times 2^{\mathcal{A}}$ *is a set of configuration states,* $k_i \in C$ *is the initial configuration state and* $\rightarrow \subseteq C \times \mathcal{R} \times C$ *is the transition relation.*

A RTS is, in essence, a labelled transition system. Transitions from each state κ are labelled with the reconfigurations that can be applied from there. States represent valid configurations of the deployed systems. Each state is actually composed of a coordination pattern; a set of state-specific constraints, which enable finer decisions (details further in Section 4.3); and a set of necessary assets for the analysis *e.g.*, PRISM specifications and symbolic port automata. Note that these models are computed in this phase in order to avoid their inherent performance overheads, later, at runtime.

4.2 The online phase: monitor feedback loop

The online phase consists of a monitor feedback loop (which springs from traditional approaches (IBM Corp 2004; Kephart and Chess 2003) built upon the reconfiguration framework presented in Section 2. Fig. 3 depicts its main elements.

We refer to this as a feedback loop based on a RTS, because the transition system of reconfigurations is a first-class entity in our approach.

Globally, our implementation of a feedback loop requires the following assets: (*i*) a RTS; (*ii*) a model of the deployed system; (*iii*) a mapper, which maps concrete connections to services to the logical ports of the model; (*iv*) the instant observations (measures) of the system properties; (*v*) a pool of candidate configurations (and their analysable assets); (*vi*) the reconfiguration framework for reasoning about the possible reconfigurations; (*vii*) the properties of interest of the system and (*viii*) the services of tools for quantitative analysis of the configuration.

Three invariants assert that (*a*) the current state (*i.e.*, the current configuration) of a RTS always points to the current configuration of the system architecture; (b) the current state of the symbolic port automata (within the current state of the RTS) reflects the current

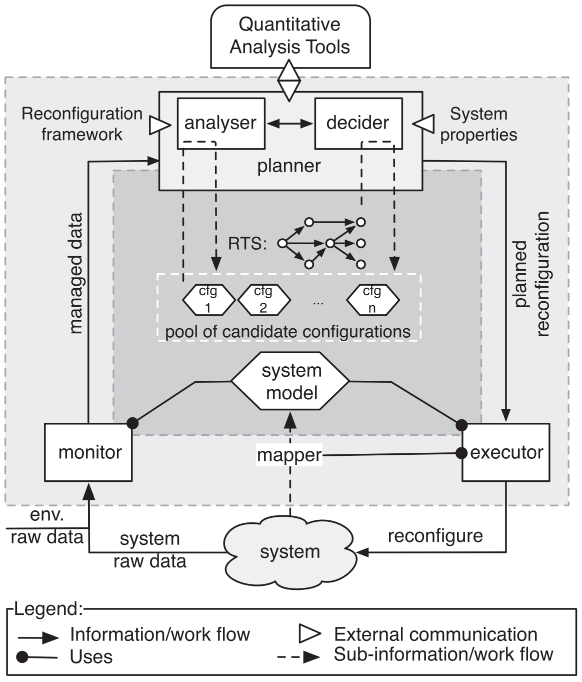


Fig. 3 Feedback loop based on a reconfiguration transition system

execution state of the system; and (*c*) the pool of candidate configurations consists of the models obtained from the current state by a single-step transition.

In the sequel we detail how the three main components (*monitor, planner* and *executor*) work together, resorting to the above mentioned assets, to achieve adaptability.

4.2.1 Monitoring

The *monitor* component aggregates data from the deployment environment and the system itself. Probes are assumed to collect different sort of data, depending on the variables that drive the adaptation. Latency, throughput, bandwidth, number of clients, number of servers or type of connection (*e.g.*, wifi, bluetooth, GSM) are typical variables. The *monitor* uses the information from the *mapper* to associate raw data from the system to the model, which is then used as-is by the *planner* component. Fig. 4 shows a UML sequence diagram which describes the interaction between these elements.

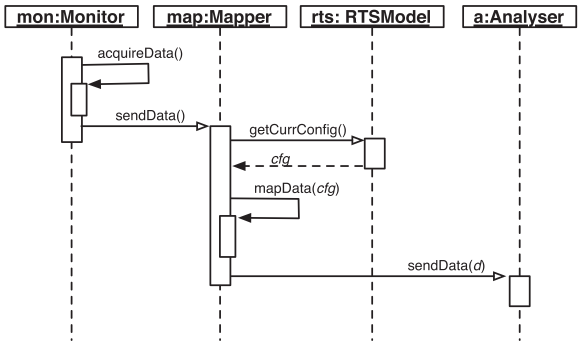


Fig. 4 Sequence diagram for the *Monitor* component

4.2.2 Planning

The *planner* has two components: the *analyser* and the *decider*, that work together to plan the most adequate adaptation to the given context. These components rely on the features of the architectural reconfiguration framework (presented in Section 2) for formally verifying the functional and non-functional properties of the architecture. Fig. 5 shows the sequence diagram for such a component. Therein, *FPChecker* and *NFPChecker* entities refer, respectively, to interfaces for the suitable functional and non-functional property analysing services.

In step marked with (1) the *decider* uses the RTS for picking all the configurations reachable from the current state. This action creates a pool of candidate configurations along with their *pre-compiled* analysable assets. In step marked with (2), the *analyser* reduces the pool by discarding configurations that fail to meet the required functional properties. These two steps are performed only once each time an adaptation occurs, or every time the functional properties of the system change.

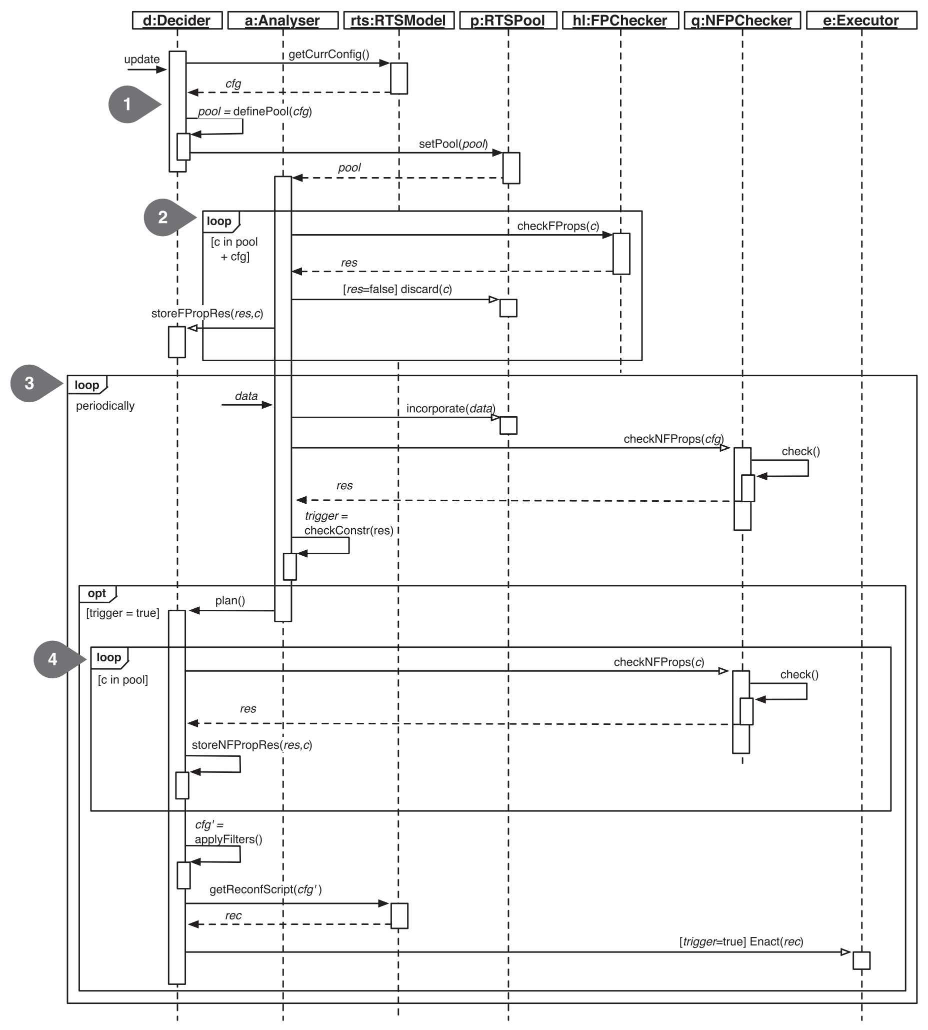


Fig. 5 Sequence diagram for the *Planner* component

Then, in the periodic loop marked with (3), the *analyser* incorporates the received managed data into the analysable assets of each configuration in the pool. This is used to check for non-functional properties of the current configuration, taking advantage of suitable quantitative analysis tools. Whenever non-functional properties fail, a reconfiguration is triggered. At this moment, in the loop marked with (4) the *decider* is responsible for choosing a suitable configuration (and associated reconfiguration operation) from the pool to embody the adaptation step. This choice, which is part of what we call the triggering of a reconfiguration, is based on the results of the (qualitative and quantitative) analyses performed.

4.2.3 Execution

The *executor* component receives the reconfiguration selected and applies it to the running system. In particular, it computes the resuming state by resorting to the symbolic port automata of the current configuration, which was derived at design time, and translates it, along with the selected reconfiguration, into an executable reconfiguration. This script is then applied to the system. This is done resorting to a *Reconfigurator* entity that is associated to the framework presented in Section 2. A *Reflector* entity, awaits for the system to reach a quiescent state; when such a state is attained, it makes the system reflect the changes by applying the reconfiguration script.

Concurrently, a sequence of updates are made: the system model is substituted by the selected configuration; the state of the RTS is updated accordingly, to meet the first feedback loop invariant; and finally, the candidate configurations in the pool are substituted by new candidates, computed in the new system's state by the decider component (*c.f.*, Fig. 5).

Figure 6 depicts the sequence diagram for the Executor component, detailing the description above.

4.3 Triggering of reconfigurations

Usually, a reconfiguration of a system is enacted whenever a non-functional property fails, violating the SLA contract. However, this vision is not always enough since the company

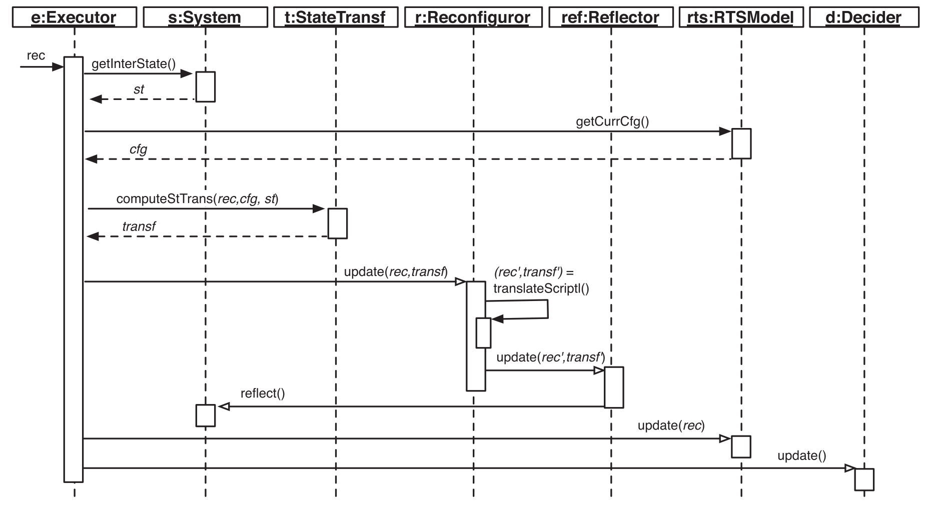


Fig. 6 Sequence diagram for the *Executor* component

owning the adaptable system may have other objectives besides providing the agreed SLA. For instance, reducing the operational costs of the system or agreeing to new functional requirements may constitute part of these objectives. Actually, in the approach proposed here, the adaptation triggering is lead by a number of constraints reflecting both the objectives of the company *w.r.t.* the system and, consequently, the adaptation logic.

Definition 5 (Trigger Constraint)**.** *Let* $c_1, \ldots, c_n \in \Xi$*. A trigger constraint is a boolean formula in disjunctive normal form,* $c_1 \wedge \ldots \wedge c_n$*.*

For instance, $(\mathsf{QoS}.p, >, 100) \wedge (\mathsf{SYS}.c, \mathsf{min}, \mathsf{true}) \wedge (\mathsf{FUN}.s, \mathsf{eval}, \mathsf{true})$ defines a trigger constraint that enacts an adaptation when the measure for the non-functional property p is not above 100, system specific property c is not the minimum (when compared to the same property of candidate configurations) and functional property s does not eval'uate to true. Prefixes are omitted when the properties provenance is clear from the context.

Once a trigger constraint is violated, the adaptation is unavoidable. But choosing a suitable new configuration is a complex task. It may even be non-deterministic or lead the system to an (infinite) chain of reconfigurations. To avoid this, it is necessary to define a base strategy to direct the choice of such configurations. Formally,

Definition 6 (Filter)**.** *Let* $c_{1_1}, \ldots, c_{1_n}, c_{2_1}, \ldots, c_{2_m}, \ldots, c_{k_1}, \ldots, c_{k_l} \in \Xi$*. A filter is a non empty, finite sequence of finite sequences*

$$\langle\langle c_1, \ldots, c_{1_n}\rangle, \langle c_{2_1}, \ldots, c_{2_m}\rangle, \ldots, \langle c_{k_1}, \ldots, c_{k_l}\rangle\rangle$$

In the sequel parenthesis are dismissed to simplify notation. The elements of a filter are separated by '|'.

A filter is used to discard, in sequence, candidate configurations that do not hold the constraint property. For example, the filter (composed of just the mandatory part) $(\mathsf{QoS}.p, >, 105), (\mathsf{QoS}.q, \mathsf{max}, \mathsf{true})$ discards, in a first step, candidate configurations that do not deliver non-functional property p above value 105 and, in a second step, it takes (from the remaining configurations) the one that delivers the maximum value for property q.

However, in some situations the filter may either discard all configurations or more than one configuration may prevail. In these cases it is possible to add optional filters to be used whenever the previous filters do not find a suitable configuration. Consider, for instance,

$$(\mathsf{QoS}.p, >, 105), (\mathsf{QoS}.q, \mathsf{max}, \mathsf{true})|(\mathsf{QoS}.p, >, 95).$$

In the case that no configuration is able to deliver a value above 105 for property p, and the second constraint is not able to pick a single configuration with a maximum value for property q, then the optional filter (the one after '|') is applied to all the pool of configurations and it will discard those that do not deliver a value above 95 for property p.

Extra optional filter elements may be added to prevent that none or more than one configuration remains. If however still multiple configurations prevail, the default is to select the first one in a ranking that contemplates the results for a prioritisation of requirements. However, for an even finer and controlled selection of a suitable configuration, constraints can be specified for each state of the RTS (*c.f.*, Definition 4). These act as specific pre-conditions to the inclusion of the corresponding configuration in the pool of candidates.

5 Application case: Adaptable-ASK

This section illustrates the application of the adaptation approach proposed in this paper to a fragment of the ASK (*Access Society's Knowledge*) system. ASK is a communication software, from the Dutch company Almende, whose objective is to mediate consumers and service providers (*e.g.* between a company looking for a temporary worker and an available person that match such a requirement). Matching mechanisms are used to combine the interveners, according to their needs (consumers) and their profiles (providers). The business goals of the ASK system are set to deliver the best consumer-provider match in the lowest time possible. This is to maximise the users' experience and their consequent return, which is the main source of revenue. On top of this, the company wants to achieve such goals while keeping the entailing costs low.

The architecture of ASK is modular, counting on three high-level components: a web-based front-end (the interface for the users), a database (that stores typical business data) and a contact engine (responsible for the matching and processing of contacts). The contact engine is the *locus* of the business: it collects the users' requests, converts them into tasks and processes them generating requests to an Executer component. Within the Executer, requests are enqueued into an Execution-Queue (EQ) until a *HandleRequest-Execution* (HRE) web-service is ready to take each one and generate *best-fit* connections between service providers and consumers. The server running the HRE service is not dedicated, but also handles other processes. Since its task of finding and establishing the best consumer-provider connection is time and resource (mainly memory) consuming, there is a top limit of 20 HRE service instances able to run concurrently. In average, each instance of the HRE service takes 0.703*s* to produce an output (*i.e.*, accepts aprox. 1.422 requests per second); this means that the server is potentially able to deal with roughly 28.440 requests per second. The EQ queue runs on a different server and is able to enqueue and dequeue at a rate of 10000 jobs per second.

The coordination model for the Executer component is as simple as shown in ❶ of Fig. 7, where *a* and *hre* are ports connecting the web interface and the HRE service, respectively; the fifo channel represents the EQ queue.

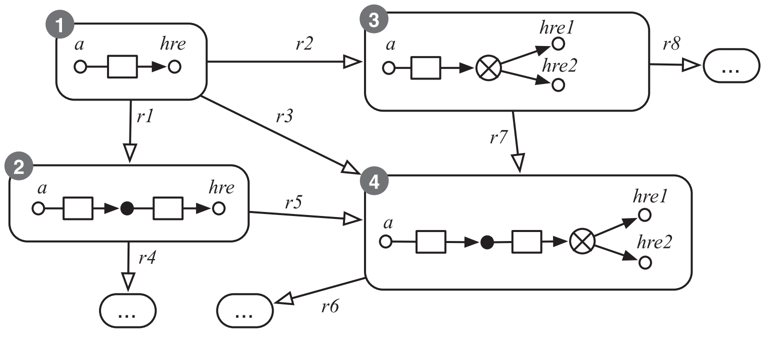


Fig. 7 RTS for the Adaptable-ASK system

The ASK system was previously studied regarding performance and resource allocation, in a static perspective (Moon et al. 2011; Moon 2011). However, the system performance fluctuates according to contextual changes. In fact, from years of experience, logs and monitored data, the ASK team has learnt that during the night there is, usually, a drop of user requests, and that after lunch until mid-afternoon, such demand reaches a peak. Moreover, it was found that roughly every six months there is a slight down time on the server where the HRE web-service is hosted. In these situations, a fixed architecture and a fixed number of resources are probably the less interesting configuration for the company. Thus, adaptation plays an important role here, in an attempt to contract the right amount of system resources and defining the most appropriate behaviour for the right environmental settings.

5.1 Planning adaptations

The context in which the ASK system operates was studied along two axis: user requests and HRE server downtimes. As already discussed, the HRE server downtimes were observed twice per year. Therefore, the rate of failure is about 6.43×10^{-8} per second[a]. Another important observation was that the mean time to recover from a failure was of about 10s. The user requests distribution by the (non-uniform) intervals of a day are depicted in Table 1.

Considering these values, it was possible to define configurations that would, most likely, overcome such changes on the environment. In Fig. 7 it is shown part of the RTS produced for the adaptation strategy of the Adaptable-ASK system.

Configuration ➊ is the original coordination pattern resorting to one queue; it has a cost per hour of €0.47. Configuration ➋ is a *scaled up* version of ➊, where more memory was added to the original queue; it has a cost per hour of €0.54. Configuration ➌ is a *scaled out* version of ➊, where a second HRE server (with same performance) is added in such a way that both servers, connected to hre_1 and hre_2, execute in parallel; this configuration has a cost per hour of €0.67. Finally, configuration ➍ is a *scaled up and out* version of ➊, where more memory and a second server are added in such a way that both servers, connected to hre_1 and hre_2, execute in parallel; it has a cost per hour of € 0.74.

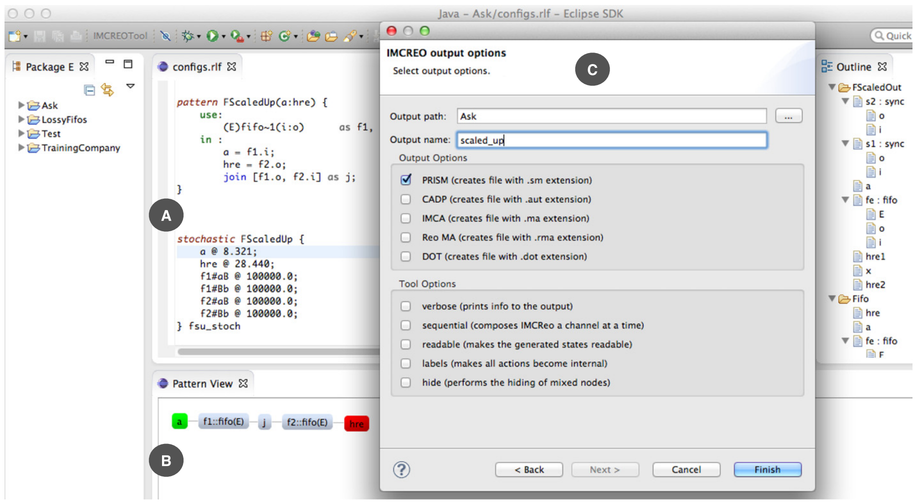
The reconfiguration operations are represented simply as r_i (for $i = 1..8$). Their concrete details are not relevant for this discussion. Also, to enhance readability, the obvious backwards reconfigurations are omitted.

5.2 Analysing RTS configurations

In a simple analysis, it is possible to see how each configuration performs against the variability of the environment data. We used CooPLa (Oliveira and Barbosa 2013a) and ReCooPLa (Rodrigues et al. 2014) languages, the associated editor and its $\mathsf{IMC}_{\mathsf{Reo}}$ tool plug-in (*c.f.*, Fig. 8) to enable such analysis. CooPLa and ReCooPLa are lightweight languages to specify coordination patterns and reconfigurations, respectively, according to the framework introduced in Section 2. Their companion editor, CooPLa editor (CooPLa

Table 1 Requests to the ASK system during a day

Hours (interval)	0–8	8–12	12–14	14–17	17–24
Requests (per second)	0.125	12.420	8.321	30.460	12.260



```
pattern FScaledUp(a:hre) {
    use:
        (E)fifo~1(i:o)      as f1,
    in :
        a = f1.i;
        hre = f2.o;
        join [f1.o, f2.i] as j;
}

stochastic FScaledUp {
    a @ 8.321;
    hre @ 28.440;
    f1#aB @ 100000.0;
    f1#Bb @ 100000.0;
    f2#aB @ 100000.0;
    f2#Bb @ 100000.0;
} fsu_stoch
```

IMCREO output options
Select output options.

Output path: Ask
Output name: scaled_up

Output Options
- [✓] PRISM (creates file with .sm extension)
- [ ] CADP (creates file with .aut extension)
- [ ] IMCA (creates file with .ma extension)
- [ ] Reo MA (creates file with .rma extension)
- [ ] DOT (creates file with .dot extension)

Tool Options
- [ ] verbose (prints info to the output)
- [ ] sequential (composes IMCReo a channel at a time)
- [ ] readable (makes the generated states readable)
- [ ] labels (makes all actions become internal)
- [ ] hide (performs the hiding of mixed nodes)

Fig. 8 The CooPLa editor and IMC$_{Reo}$ tool plugin. A - the main text editor with CooPLa specification of the *scaled up* configuration (➋) and its stochastic instance; B - the view for the visualisation of the pattern under edition; C - the IMC$_{Reo}$ tool wizard, to convert the configuration into a PRISM model

Team 2014), is an Eclipse plug-in with features for edition time code completion, semantic suggestions and visualisation of coordination patterns. The IMC$_{Reo}$ tool is a plug-in of the editor that converts coordination patterns into IMC$_{Reo}$ models (Oliveira et al. 2014, 2015), which can than be converted to inputs for a range of well known quantitative analysis tools. PRISM was used in this case-study to verify the quantitative properties asserted on each configuration.

A property of interest for the ASK team is the *throughput ratio* (TR) for the long run. This is, the ratio between the effective throughput and the maximum throughput possible. In PRISM, such a property can be formulated using the notion of *rewards* as follows: `R{"runs"}=? [S] / T`, where `runs` is a reward structure that assigns the value 1 to each transition that transmits data to hre_1 (and hre_2); and `T` is a variable representing the user requests. Table 2 summarises the values obtained for this property at the precise rate of user requests assigned to each hour interval.

The non-faulty server (NFS) and faulty-server (FS) marks are relative to experiments where, in the first one, the server connected to port hre_1 was always available and, in the second, was constantly failing (accepting one request in each 10s); in both cases, the

Table 2 Steady-state throughput ratio analysis for the several hour intervals

Hours (interval)		0–8	8–12	12–14	14–17	17–24
Requests (per second)		0.125	12.420	8.321	30.460	12.260
➊ Original	NFS	0.999	0.950	0.981	0.721	0.951
	FS	0.661	0.008	0.012	0.003	0.008
➋ Scaled Up	NFS	0.999	0.978	0.994	0.769	0.979
	FS	0.702	0.008	0.012	0.003	0.008
➌ Scaled Out	NFS	0.999	0.996	0.998	0.944	0.996
	FS	0.999	0.951	0.982	0.722	0.952
➍ Scaled Up and Out	NFS	0.999	0.998	0.999	0.970	0.998
	FS	0.999	0.978	0.994	0.770	0.979

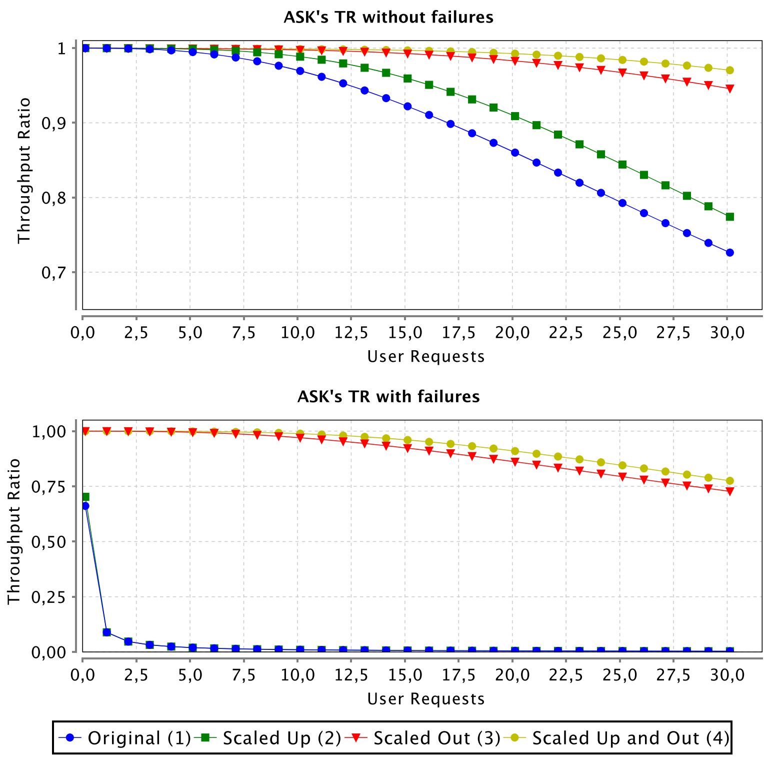


Fig. 9 Performance analysis of the throughput ratio property for the several configurations. Without faulty server (above) with faulty server (below)

server on port *hre*$_2$ (when present) was always up. The graphs in Fig. 9 provide a similar view, but now depicting an evolution of the TR property depending on the number of user requests (which vary from 0 to 30 requests per unit of time). The upper graph shows the evolution of TR for the servers without failures; the bottom one shows the same evolution considering the the presence of a faulty server, in the conditions explained before.

5.3 Predicting adaptations by objectives, constraints and filters

Adding resources like servers and memory to the system is costly as shown by the cost per hour indicated for each configuration. Assuming that these resources are *paid-per-use* as in a cloud environment, it is essential to spend only the minimum required time on the proposed configurations.

But delivering a service only with minimum costs in mind is not advantageous, since the obvious slowlyness of the system will alienate its customers. This brings the need for defining a suitable service level agreement (SLA) for the system. As such, the ASK team defined that an optimal value for the TR QoS property would be above 0.970[b] (in the sequel 0.970 is referred to as TR threshold, or t for short).

This being fixed, the ASK team defined then two important properties for Adaptable-ASK: QoS.TR and SYS.cost, and based on them the following trigger constraint:

$$(TR, \geq, t) \wedge (cost, \text{min}, \text{true})$$

Table 3 associates the most suitable configuration to each hour interval, considering multiple adaptation objectives, defined by suitable filters.

Table 3 Predicted configurations for each hour interval and associated triggering filters

	0–8	8–12	12–14	14–17	17–24
(cost,min, true)	❶	❶	❶	❶	❶
(TR,max, true)	❶	❹	❹	❹	❹
NFS – (TR,$\geq$, *t*),(cost,min,true)	❶	❷	❶	❹	❷
FS – (TR,$\geq$, *t*),(cost,min,true)	❸	❹	❸	??	❹
NFS – (TR,$\geq$, *t*),(cost,min,true) (TR,max,true)	❶	❷	❶	❹	❷
FS – (TR,$\geq$, *t*),(cost,min,true) (TR,max,true)	❸	❹	❸	❹	❹

The top two rows are concerned with the selection of candidate configurations filtering, exclusively, by minimum cost and maximum TR value, respectively. As expected, these filters define adaptation strategies that make the system practically fixed. The top one reduces company costs, but also the TR values; the second augments the TR value (by increasing customer satisfaction), but at higher costs. The third row presents a filter that selects first the configurations delivering a TR value above the SLA threshold, and then selects the one with minimum cost. For the NFS setting (*i.e.*, all the servers are up), the selected configurations are balanced and thus, the adaptation is more in line with the company objectives. In a situation FS (*i.e.*, one server is continuously failing), however, there is no configuration able to deliver a TR above the desired threshold for the interval where the user requests reach a peak (*i.e.*, 14–17). In this case, the system would not reconfigure itself. If for some reason the active configuration at that moment is ❶ or ❷, then the system would perform low (see Fig. 9, bottom graph) for a while, increasing the losses for the company. On the other hand, the fourth row extends the previous filter by adding an optional filter that selects the configuration delivering the maximum TR value, when the first filter is not able to propose a configuration. Therefore, it is now possible to have a suitable configuration for situation *ii*) when the users demand is higher.

Since the last filter provides a balanced adaptation strategy it was chosen by the ASK team as the runtime filter that ensures the company objectives.

5.4 A runtime situation

At runtime, however, the environment changes are more continual and unpredictable. Therefore, the previous analysis and the adaptation strategy form only a basis for what must be finely tuned at runtime. In any case, the more accurately the analysis in the offline phase is, the better the results in the online phase will be.

Since the dynamic part of the adaptation methodology proposed here is not currently implemented, we used simulation to predict how the defined adaptation strategy for the Adaptable-ASK system would behave in a real runtime situation.

Thus, the system's execution for one day was simulated. It was assumed that servers will not fail along this period; and the user requests will be obtained from traces of the system, such that the average in each part is the one shown in Table 1. The results of the simulation are given in Fig. 10. Performance was evaluated at each minute, considering the current request rate and the four configurations: the active one and the three candidates. The exception is when the active configuration is ❷ or ❸, for which the candidates are only configurations ❶ and ❹[c].

From the top graph in Fig. 10, we see that the first need for adaptation occurred at minute 480, which means that for the first 8 hours of the day, the system has shown a

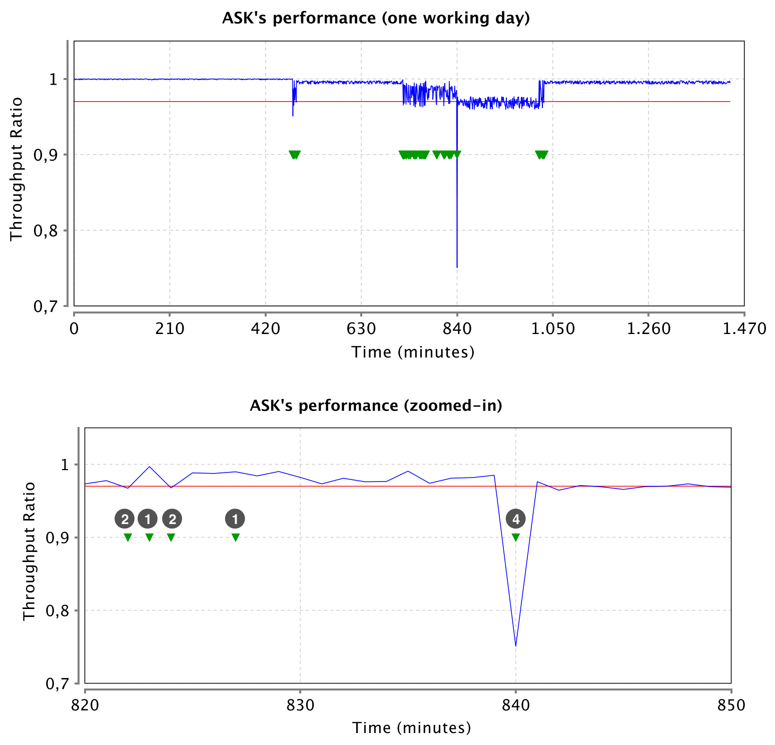


Fig. 10 Performance of adaptable ASK system. For one working day (above); and a zoomed view of concrete configurations (below)

good performance while in configuration ❶. Then, in the first minutes of the 8th hour of execution, the system adapts until stabilised for the amount of requests. However, from minute 720 until minute 840 the system is constantly adapting itself. Three hours later, at minute 1020, the system adapts again for some times until it stabilises for the rest of the day. In the bottom graph of Fig. 10, we zoomed-in in a zone that spans for 20 minutes before entering the peak of requests (at minute 840) and 10 minutes during it. Before entering the peak zone, the system is able to deal with the requests in its original configuration: ❶. Notice that the second adaptation to configuration ❶ is enacted not because the system is performing below the TR threshold, but because there is a cheaper configuration that delivers similar performance. This is the intended behaviour as requested by the ASK team. However, when the users' requests augment significantly, the system performs below the TR threshold and therefore adapts to configuration ❹. In the subsequent minutes there are no adaptations even though the system performs roughly below the TR threshold. This is because (*i*) there are no selectable configurations after filtering and (*ii*), the alternative filter (TR,max, true) defined for the adaptation strategy keeps selecting configuration ❹.

5.5 Discussion

In this simulation, along 24 hours the system adapted 48 times, with a *mean time to adapt* of 1800s (*i.e.*, 30 minutes)[d]. Although this seems to be a reasonable value, it may be misleading. In fact, notice that the system only adapts itself in, roughly, three parts of the day; the most critical one spanning from minute 720 to minute 840, where 75% of adaptations occur (a local mean time to adapt of 200s, or roughly 3.3 minutes). This increases the time spent in reconfigurations (for simplicity we assumed them to be instantaneous), which consequently decreases the productivity of the system.

Such a situation can be mitigated by increasing the complexity of the adaptation algorithm, namely in what concerns analysis and decision. For instance, instead of choosing a configuration based on its performance on the current rate of requests, we could use the history of requests (or at least the last n rates) to predict the next one, and elaborate the decision based on the system performance for such a prediction. Also, we could resort to some notion of *hysteresis* to gracefully stabilise the system. For instance, this could delay the next adaptation for some time or until a cheaper configuration does not ensure a TR value above some threshold $X > 0.970$. The latter would improve performance and, in the long run, decrease the costs (that may be associated to reconfigurations).

From an economical point of view, the simulation has shown that the company would pay around € 11 per day for the system configurations and resources used. This value, compared with the one obtained if constantly using the most expensive configuration (around € 18 per day), shows that the adoption of this strategy would make the company save about € 2500 per year. While it is not a huge value, it shows that there are benefits on using this approach. Further refinements on the RTS and its constraints have potential to improve the savings.

In order to keep the example simple and understandable, the coordination patterns considered in Fig. 7 are simple and omit several parts of the coordination of the whole ASK system. We deliberately set aside the use of structural properties to define system functional requirements to be preserved during the adaptation, and which could be used to rule out some candidate configurations. Moreover, being a simulation, this example has left state transfer out of the equation. The strategy for consistent state transfer subsumes imperceptible computational efforts within the whole adaptation strategy. Thus it would not affect the obtained results.

The symbolic port automata in Fig. 2 are similar (up to port and state names) to those underlying the configurations considered in the example: the top one corresponds to configurations ❷ and ❹; the bottom one corresponds to configurations ❶ and ❸. For instance, the state transfer computed for Fig. 2 would also apply in a reconfiguration from configuration ❶ to ❷.

6 Adaptation as a Service

The self-adaptation strategy approach we propose in this paper can be reused in different systems since only its central pieces (properties, constraints, filters and the RTS) are system-dependent. This assures the so desired separation of concerns between managed and managing systems (Weyns et al. 2013). Such a separation is not a novelty. Most self-adaptation approaches promote it (Garlan et al. 2004); and the MAPE-K reference model almost obliges it. However, notwithstanding the separation of concerns, managed and managing systems are usually running in the same physical execution environment. This makes the managed system to decrease its performance, since the feedback loop allocates part of the available resources for its own use.

A possible solution for such a problem is to physically separate both entities. This entails the need for companies to acquire more processing and storage power as well as to be willing to manage such extra resources with all the costs associated. A smoother solution is to rent virtual machines from a cloud service, and deploy therein the feedback loop system. On the one hand, this eases management, but on the other hand it requires an extra effort in order to set the whole system up.

In order to avoid these problems, we propose a new strategy towards delivering adaptation as a service (AaaS).

6.1 Architecture and main workflow

The essential components of our feedback loop (monitor, analyser, decider and executor) are loosely coupled entities with a specific behaviour. Regarding them as services is therefore natural. With this in mind, we propose to refactor our self-adaptation strategy in Section 4, so that the essential parts of the feedback loop are deployed in the cloud for immediate usage. The expected result is that the common computational activities for adaptation (*e.g.*, analysing data for perceiving the need for adaptation or deciding which reconfiguration to choose among a set of possible ones) are transparent to (and not developed by) the users. Fig. 11 presents the overview of the expected global architecture, along with traces of the main workflow for both users and the adaptation service. In the next paragraphs, the *adaptation service* will be referred to as AaaS, and the hosting cloud as AaaS cloud.

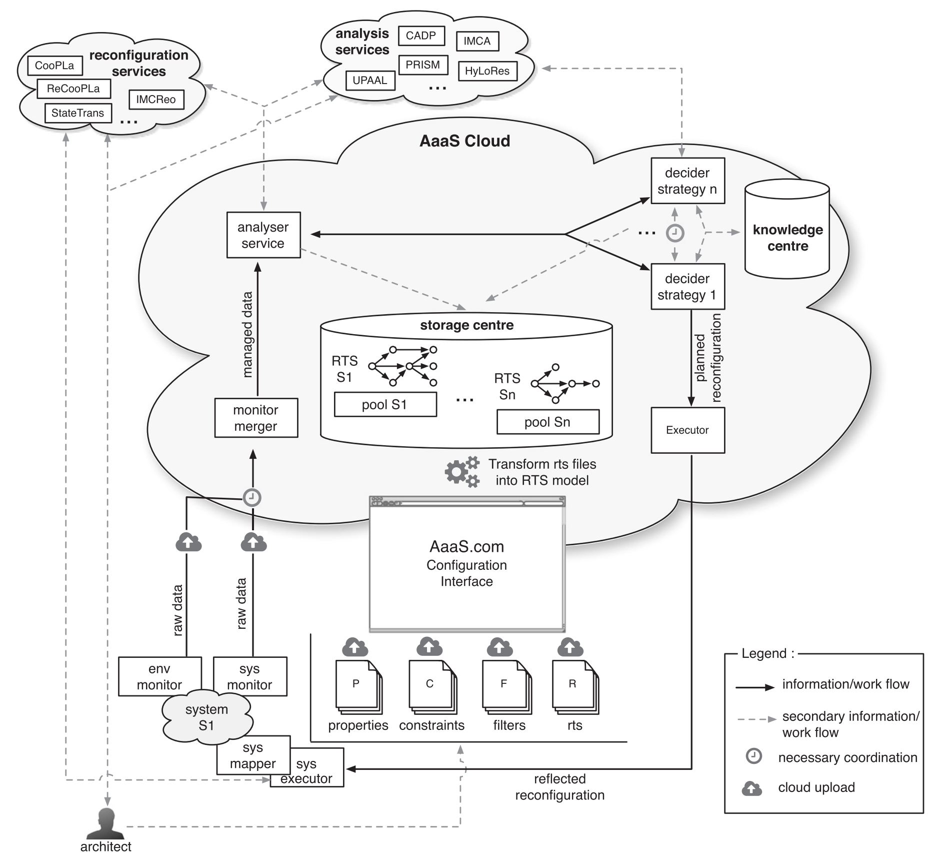


Fig. 11 Adaptation as a Service architecture overview

Actually, we take all the tasks that are known to be time and resource consuming, and encapsulate them as services. In particular, we assume the existence of online versions of established analysis tools (*e.g.*, CADP, PRISM, IMCA, HyLoRes, among others) that make available, through public interfaces, services of which AaaS will be client. Moreover, the tools associated to the formal framework presented in Section 2 are also assumed to be available as services in a dedicated cloud environment. These two sets of services are expected to release most of the workload from the feedback loop.

The feedback loop constitutes the core of the AaaS. We made it more comprehensive by supporting multiple monitoring and decider components, in an attempt of decentralising the feedback loop (André et al. 2011; Nallur and Bahsoon 2013; Vromant et al. 2011; Weyns et al. 2013). This comes with the price of extra coordination and synchronisation effort. But it is essential. For instance, instead of having a single filter-based strategy to decide reconfigurations, we can have several others, including one that uses artificial intelligence techniques (*e.g.*, case-based reasoning) to make such a decision. The results of all decider components have to be coordinated. Only one will prevail, but such a decision will be endowed with extra robustness.

AaaS is able to track more than one single system. The cloud support for multi-tenancy and the service-orientation of the approach allow AaaS to deliver the same adaptation service with the same expected quality to several systems. For this, each tracked system is given a space in a storage centre, where the RTS and the current pool of configurations are placed. AaaS remains loyal to the coordination-centred vision for reconfigurations, though. Moreover, in Section 4, we assumed that the managed systems could be distributed but their coordination layer had to be centralised. With a large-scale approach like AaaS, that assumption makes no sense. Thus, although there must be a main coordination entity for a distributed system, there can be several sub-coordination entities distributed in several nodes of the same system that are themselves tracked by AaaS. Again, this is based on the theories for feedback loop decentralisation discussed by D. Weyns et al. 2011, 2013, and consequently, requires a distributed notion of coordination-targeted reconfigurations (Koehler et al. 2009), which is out of the scope of this paper.

In the sequel we exploit the offline and online phases of this cloud based approach for system adaptation.

6.1.1 The offline phase

In this phase the architects have to prepare the assets (as suitable files) that make adaptation possible. This includes the system properties, that translate functional and non functional requirements; the constraints, that define the system goals for adaptation; the filters, that define the main strategy for deciding the reconfiguration to lead the system into a desired configuration; and finally the RTS.

The production of the RTS is a complex and time-consuming task. To help the architects accomplishing it, the reconfiguration services assumed can be used; in particular, the $\mathsf{IMC}_{\mathsf{Reo}}$ translation service, which becomes computationally heavy as the complexity of the system coordination patterns grow. The analysis tools to fine tuning thresholds and other measures are also assumed to be used from the available services. In the end, the RTS is expected to be delivered as a comprehensive set of files written in CooPLa (for the definition of coordination patterns) and ReCooPLa (for the reconfiguration scripts). Together with the other assets, all these files have to be uploaded to the AaaS cloud through the configuration interface as depicted in Fig. 11. Once uploaded, the RTS files are transformed into a RTS model and all the associated assets (*e.g.*, the final PRISM files) of each state are conveniently generated and stored in the storage centre.

The configuration interface is expected to guide the architect through all the configuration of an instance of AaaS. Besides the upload of the required files, the architect is also able to choose, for instance, which analysis tool(s) shall be used to verify the properties of the system or which strategy(ies) shall be applied to decide the reconfigurations to apply when needed.

In addition, the architect is responsible for coupling monitors to systems that are able to ship data to the AaaS cloud every time a (relevant) change occurs either in the environment or internally. The architect has also to define a local mapper component that contributes a reflection model of the managed system. A local executor component, actually an AaaS off-the-shelf component, also needs to be attached to the system.

6.1.2 The online phase

When the configuration is over and the architect decides to explicitly enable AaaS to manage its system, the online phase begins.

As expected, monitors ship data to the AaaS cloud, which is synchronised and merged therein. A monitor merger service is assumed to merge the monitored data and send it to the analyser service. The latter behaves exactly as before. The particularity is that it now evokes services for the necessary quantitative analysis. It is still responsible for triggering the need for a reconfiguration by analysis of the user-uploaded constraints.

When an adaptation is triggered, the decider (or deciders) start the analysis to plan a new adaptation. Depending on the user configuration, one or more strategies may be associated to the managed system. Each strategy is different. For instance, the filter-based strategy uses the analysis services to analyse the configurations in the pool, which are sent as a unique workload. A strategy adopting case-based reasoning mechanisms would delegate its tasks into services to that end, but will rely on a knowledge centre to define its decision, as depicted in Fig. 11. The decider service is also responsible for updating the pool of configurations, as explained in Section 4.

Upon decision, the chosen reconfiguration is passed to the executor. The executor translates the reconfiguration into a script able to concretely apply the changes to the managed system. This script is passed to the local executor component. The latter uses the reconfiguration services to compute the resuming state, and when the system enters a quiescent state, applies the changes via the mapper component. The option of having a local executor component is due to AaaS being not aware of the internal state of the systems it manages. Thus, interrupted and resuming states have to be computed locally. This is also necessary because such states have to be computed in the instant before the changes are applied to the system, so that the managed system consistently resumes its production.

6.2 Discussion

The AaaS approach brings several benefits when compared to traditional approaches. It promotes a clear (physical) separation between the managed system and its feedback loop. It allows architects and developers to focus on the design and development of the system and, consequently, frees them from dealing with the always complex implementation of feedback loop components. It eases the evolution of legacy static systems into self-adaptable ones and allows for more comprehensive and robust decisions, by enabling the combination of several strategies. Moreover, it enables the decentralisation of the feedback loop, augmenting the dependability of the system as a whole.

AaaS is a one-size-fits-all approach for adaptation. This can be seen as a drawback, but in fact the approach is highly configurable in order to support the demands of their tenant systems. In fact, the adaptation logic is mainly delivered by the architects in the uploaded analysable assets. AaaS essentially performs intense computations in order to deliver decisions based on such assets. The adaptation logic is not static. At any time the company may change its goals or the system requirements, or the architects may update the RTS to cope with new system configurations. This entails the need for re-uploading new asset files. The AaaS is expected to reconfigure its behaviour to conform to these changes immediately. Moreover, the customisation of AaaS behaviour can be performed at any time, as well.

Although AaaS service is configured by the architects, behind the scenes, a local feedback loop will ensure the correct work of each AaaS instance monitoring each client system. This will enable necessary adaptations when, for instance, some component of an AaaS instance fails to respond or when the AaaS infrastructure needs to enlarge its computational power for continuously ensuring correct load balancing.

The approach, however, has limited applicability in time critical software systems or in application highly distributed by mobile devices. In the first case latency may impair real timeliness. In the second, because mobile networks are often unstable.

However, surveillance systems, asynchronous communication systems (like ASK) and many others that may adapt to context changes but are not time critical, would benefit from such an infrastructure. Usually deployed in environments with a stable network infrastructure, these systems are able to exchange data with the remote servers of AaaS, and perform the necessary computation for adaptability.

7 Related work

Our proposal of a self-adaptation strategy mainly focuses on two aspects. The first one concerns the reconfigurations of the coordination layer and their planning/organisation in a relational structure. The second one is the integration of a formal framework in a feedback loop, allowing for detecting, deciding and triggering adaptations.

Several approaches to implement feedback loops for self-adaptive systems are reported in the literature. In general, these approaches agree on external, reusable and component-based feedback loops implementations, rather than on internal, monolithic, and intertwined implementations (Cheng et al. 2009; Huebscher and McCann 2008; Salehie and Tahvildari 2009). How adaptations are decided and which assets are used to aid in such decisions differ from case to case. In the sequel we compare our approach with other works along three dimensions: *i)* quantitative analysis; *ii)* design, detection and selection of adaptations; and *iii)* the use of models as system-knowledge artefacts for the feedback loop implementation.

7.1 Adaptations and quantitative analysis

In (Calinescu and Kwiatkowska 2009; Calinescu et al. 2012), the authors present a framework for the adaptation of software systems, where system components are modelled as Markov chains. The framework takes advantage of quantitative model checking, using PRISM, to analyse the components and dynamically adjust the system to its objectives and the changes in the environment. Specific policies are used to define constraints to which the system should agree or measures of success that it must optimise. Adaptations are made on the configurable parameters of the system that realise the policies. Our approach shares with this one the use of quantitative model checkers (*e.g.*, PRISM) to analyse the system. However, we focus on the coordination layer and use (interactive) Markov chains to analyse it, rather than the components themselves. Moreover, the adaptations we assume are made to the structure of the coordination and not to the parameters of the system.

Another approach, documented in reference (Becker et al. 2013) is based on simulation of a specific-modelled system to gradually find a suitable point to trigger adaptations and consequently to fulfil system requirements. This is done for a range of possible (static) contexts and through multiple design iterations. Similarly, we analyse possible system coordination configurations, but instead of proposing a single design, we propose a relational structure that captures several designs, which are likely to perform well against contextual variability. Tools for performance analysis (Becker et al. 2009; Bondarev et al. 2006; Grassi et al. 2009, 2007) that take into account the performance of the original system and may integrate part of a feedback loop component to enact adaptations, should also be mentioned.

7.2 Languages for adaptation specification

In (Huber et al. 2014) the authors propose the S/T/A domain-specific modelling language to describe runtime adaptation processes on top of QoS models of component-based system architectures. S/T/A is used to define strategies, tactics and actions for adaptations. Strategies define system goals; tactics define how to proceed on an adaptation; and actions are the atomic elements that change the system configuration. Weights are assigned to tactics, after simulation, to define the impact of applying them to the running system. Then, they can be used by strategies to determine which tactic to apply next. Our approach also defines strategies (referred to as trigger constraints), tactics (reconfigurations) and actions (which are seen as primitive reconfiguration operations). Differently from this approach, we do not base the choice of a reconfiguration only on its impact. Instead, we concretely define filters to select the most appropriate reconfiguration for the current environment settings.

Reference (Cheng and Garlan 2012) introduces Stitch, a language to define strategies and tactics. Each tactic defines a condition for its applicability, a set of actions (that apply changes to the system) and a set of effects, which may be regarded as adaptation post-conditions. A strategy encapsulates an adaptation processes, by using tactics in a deterministic *if-then* approach. In this paper we do not present a language to concretely define the triggering of adaptations and the selection of the most appropriate reconfiguration. However, we define constraints and filters, which detects and enacts adaptations based on the general objectives of the system and not on specific events. Other languages like Acme (Garlan et al. 1997), Wright (Allen 1997) or YAWL (van der Aalst and ter Hofstede 2005) allow for the specification of adaptations. However, their specific use as ADLs or workflow languages, limit their use in the specification of proper adaptation strategies.

7.3 Models in adaptation approaches

Models are used extensively as part of feedback loop implementation strategies. They usually convey the architecture of a running system at a level of abstraction suitable for analysis. In (Garlan et al. 2004), notions of architecture style, invariants, operators and properties are used to define strategies of adaptation, where invariants are checked upon a model of the system that is seen as an abstract graph of computational elements, upon which behaviour and specific properties are defined.

In (Litoiu et al. 2008), the authors propose an adaptation strategy to guarantee web services quality. In particular, they propose a control loop implementation that is based on a model of the web service and a robust estimator, used to keep the QoS values in accordance to the SLA. In (Floch et al. 2013; Hallsteinsen et al. 2012), MUSIC is presented as a framework for model driven development of (component and SOA-based) adaptable mobile applications in the context of ubiquitous computing. It relies on models of the context and of the application architecture; the latter being annotated with application adaptation capabilities and its dependencies to the context. Moreover, it instantiates the MAPE-K architecture and uses a reasoner to search the set of possible configurations for the optimal solution in the current context. When an adaptation is required, a reconfiguration script is derived and executed. How the best configuration is determined, concerning QoS and the SLA, is not clearly reported, however. In (Agrawal et al. 2003; Fischer et al. 2000), UML is used along with graph transformation techniques to define the adaptation of systems. In this approach, performance analysis is not natural, but checking behavioural and structural properties bedomes easier using constraint languages like OCL. Nevertheless, all of these approaches use *ad hoc* models. Our approach, on the other hand, resorts to a generic graph-based model that may borrow structure and behaviour from formal models like Reo, later transformed into (interactive) Markov chains. Also, to the best of our knowledge, this is the first attempt of issuing a self-adaptation strategy for software systems with the focus on the analysis and adaptation of the coordination layer that leads the global system architecture.

7.4 Decentralised self-adaptation

Decentralised approaches for self-adaptation use several feedback loops (or several of its components) to control a system (typically complex and distributed) (Vromant et al. 2011; Weyns et al. 2013).

In (Caprarescu and Petcu 2009) the authors, inspired from natural adaptive systems, propose a robust feedback loop for computational systems. Multi-agent technology and swarm intelligence is used to define decentralised feedback loops that mimic ant colonies. The authors stress that use of multiple feedback loop agents enables robustness, for when one agent fails, the others may continue by enacting self-organisation.

In reference (André et al. 2011) it is proposed a framework (SAFDIS) for adaptation of distributed service-based applications, that is fully decentralised. Feedback loops are regarded as independent applications, external to the managed systems which they control. Each such loop adapts the associated members of the distributed managed application. Cooperation via coordination and negotiation is, however, part of the decision making algorithm. Moreover, SAFDIS is implemented as a SOA system, enabling its components to be used as services by developers and architects.

MOSES (Cardellini et al. 2012) is proposed as a methodology to support QoS-driven adaptation of service-oriented systems. In particular, it acts as a service broker in order to provide the best selection and binding of services for a suitable description of the system architecture and its companion non-functional requirements. Decentralisation occurs at the monitor level. Several monitors collect data about QoS of regionally distributed pools of services, that are candidates for binding to the managed system. The remaining tasks of the MAPE-K reference model are centralised, though.

In (Nallur and Bahsoon 2013) the authors focus on a market-oriented programming strategy to define adaptation strategies. They consider several market places where seller services offer their QoS attributes for some cost, and buyer applications bid for services with a desired QoS and the price willing to pay for such service. Markets, as distributed places, make the approach decentralised, since several decider agents have to work in each market for a suitable solution.

Although SAFDIS framework (André et al. 2011) being, however, close to our proposal, none of the above decentralised approaches for feedback loops intends to deliver adaptation as a service.

8 Conclusions

The paper discussed an architectural adaptation strategy for systems able to self-adapt in accordance to the surrounding environment. It is based on two phases: one offline, where reconfigurations are planned and organised; and another online, that takes advantage of such organisation of reconfigurations to autonomously choose one and adapt the system, as part of a monitoring feedback loop. This strategy acts on top of a concrete framework that allows the software architect to model and apply reconfigurations and to formally verify and reason about functional and non-functional (quantitative/probabilistic) requirements of the system architecture. We highlight the use of formal models to represent the coordination layer of a software system. Through source-to-source transformation techniques these models are transformed into suitable quantitative models, enabling runtime verification of both non-functional and functional requirements of the system. This plays a crucial role in triggering adaptations, and, in general, in the maintenance of software architecture quality, and system consistency upon dynamic reconfigurations.

The use of formal methods, in contrast to other approaches commonly employed by practitioners (*e.g.* UML, rule-based, etc.) allows for a precise specification of patterns, reconfigurations and properties, as well as their verification through appropriate tools. A slighter heavy, and certainly less usual notation is a price to be paid. Nevertheless both the CooPLa (Oliveira and Barbosa 2013a) and ReCooPLa (Rodrigues et al. 2014) editors, that support architectural design in this framework, and the plugged-in verifiers, have user-friendly interfaces and are relatively easy to use. In any case there is no alternative in Software Engineering to the road towards increased precision and rigour.

Based on this adaptation architecture, the paper also proposes a cloud-based implementation of a feedback loop that is transparent to the users and delivers adaptation as a service. Among several advantages, we highlight the fact that it frees the users to actually develop such a feedback loop, and gives them total control of how the system shall evolve in each situation, by enabling a fully configurable cloud environment.

Currently, we are developing a prototype implementation of the approach introduced in Section 4 on top of the reconfiguration framework mentioned in Section 2. Moreover, we are studying how the RTS model can be delivered as a weighted automata where the edges are labelled with reconfigurations and their application costs. Weighted automata theory would allow for addressing overall reconfiguration properties like, for instance, "*in one year, the overall time spent on reconfigurations shall remain below 120s*".

A complex problem is still to solve, though. As pointed out in the SBCARS'2014 session, we are naively assuming that each RTS state has a small number of transitions. Scalability issues would arise if that number grows bigger; meaning more configurations in the pool and consequently more time doing heavy quantitative analysis and arriving to a decision. A possible solution will resort to RTS-specific bisimulation techniques in order to minimise that structure.

Endnotes

[a]The *mean time between failure* (MTBF) QoS attribute of the server is consequently set to $15552000s = (360 * 24 * 60 * 60)/2$.

[b]For the purposes of this paper, the SLA of the ASK system is comprised only of this TR property and its derivates, like throughput, latency or response time.

[c]Remember that inverse reconfigurations are omitted in the RTS of Fig. 7 but assumed to exist.

[d]Notice that reconfigurations were assumed to take effect in a negligible (near to instantaneous) amount of time.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
NO developed the adaptation strategies with all associated techniques, designed and conducted the case study and drafted the manuscript. LSB discussed the obtained results and thoroughly revised the paper. All authors read and approved the final manuscript.

Acknowledgements
We would like to thank the SBCARS'2014 reviewers and conference participants for the questions raised, which we have tried to address here. This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through FCT, the Portuguese Foundation for Science and Technology, within project FCOMP-01-0124-FEDER-028923. Author Nuno Oliveira was supported by a Doctoral Grant from FCT, with reference SFRH/BD/71475/2010.

Received: 4 December 2014 Accepted: 11 May 2015
Published online: 28 May 2015

References
Agrawal A, Karsai G, Shi F (2003) A UML-based graph transformation approach for implementing domain-specific model transformations. Int J Softw Syst Modeling1–19
Allen R (1997) A formal approach to software architecture. PhD thesis, Carnegie Mellon, School of Computer Science, Pittsburgh, PA, USA. (January 1997). CMU Technical Report CMU-CS-97-144
André F, Daubert E, Gauvrit G (2011) Distribution and self-adaptation of a framework for dynamic adaptation of services. In: The Sixth International Conference on Internet and Web Applications and Services (ICIW). IARIA, Red Hook, NY, USA. pp 16–21
Arbab F (2004) Reo: A channel-based coordination model for component composition. Math Struct Comp Sci 14(3):329–366
Basu A, Bensalem S, Bozga M, Combaz J, Jaber M, Nguyen TH, Sifakis J (2011) Rigorous Component-Based system design using the BIP framework. Software IEEE 28(3):41–48
Becker M, Luckey M, Becker S (2013) Performance analysis of self-adaptive systems for requirements validation at design-time. In: Proceedings of the 9th QoSA '13. ACM, New York, NY, USA. pp 43–52
Becker S, Koziolek H, Reussner R (2009) The palladio component model for model-driven performance prediction. J Syst Softw 82(1):3–22
Blackburn P (2000) Representation, reasoning, and relational structures: a hybrid logic manifesto. Logic J IGPL 8(3):339–365
Blackburn P, de Rijke M, Venema Y (2001) Modal Logic. Cambridge Tracts in Theoretical Computer Science (53). Cambridge University Press, Cambridge

Bondarev E, Chaudron M, With P (2006) A process for resolving performance Trade-Offs in Component-Based architectures. In: Component-Based Software Engineering. Lecture Notes in Computer Science, vol. 4063. Springer, Berlin, Heidelberg. pp 254–269

Bonsangue M, Clarke D, Silva A (2012) A model of context-dependent component connectors. Science Comput Programm 77(6):685–706

Brauner T (2010) Hybrid Logic and Its Proof-Theory. Applied Logic Series. Springer, Berlin, Heidelberg

Brun Y, Serugendo GM, Gacek C, Giese H, Kienle H, Litoiu M, Müller H, Pezzè M, Shaw M (2009) Engineering Self-Adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin, Heidelberg. pp 48–70

Calinescu R, Kwiatkowska M (2009) Using quantitative analysis to implement autonomic IT systems. In: Proceedings of ICSE'09. IEEE Computer Society, Washington, DC, USA. pp 100–110

Calinescu R, Ghezzi C, Kwiatkowska M, Mirandola R (2012) Self-adaptive software needs quantitative verification at runtime. Commun ACM 55(9):69–77

Caprarescu BA, Petcu D (2009) A Self-Organizing feedback loop for autonomic computing. In: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:. IEEE Computer Society, Washington, DC, USA. pp 126–131

Cardellini V, Casalicchio E, Grassi V, Iannucci S, Lo Presti F, Mirandola R (2012) MOSES: A framework for QoS driven runtime adaptation of Service-Oriented systems. IEEE Trans Softw Eng 38(5):1138–1159

Cheng BH, Lemos R, Giese H, Inverardi P, Magee J (2009) Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin, Heidelberg. pp 1–26

Cheng SW, Garlan D (2012) Stitch: A language for architecture-based self-adaptation. J Syst Softw 85(12):2860–2875

Ciraci S, van den Broek P (2006) Evolvability as a quality attribute of software architectures. In: Proceedings of the International ERCIM Workshop on Software Evolution. UMH, Mons. pp 29–31

Dobson S, Denazis S, Fernández A, Gaïti D, Gelenbe E, Massacci F, Nixon P, Saffre F, Schmidt N, Zambonelli F (2006) A survey of autonomic communications. ACM Trans Auton Adapt Syst 1(2):223–259

Fischer T, Niere J, Torunski L, Zündorf A (2000) Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In: Theory and Application of Graph Transformations. Lecture Notes in Computer Science, vol 1764. Springer, Berlin, Heidelberg. pp 296–309. Chap. 21

Floch J, Frà C, Fricke R, Geihs K, Wagner M, Lorenzo J, Soladana E, Mehlhase S, Paspallis N, Rahnama H, Ruiz PA, Scholz U (2013) Playing MUSIC – building context-aware and self-adaptive mobile applications. SPE 43(3):359–388

Garavel H, Lang F, Mateescu R, Serwe W (2012) CADP 2011: a toolbox for the construction and analysis of distributed processes. Int J Softw Tools Technol Transfer 15(2):89–107

Garlan D, Monroe RT, Wile D (1997) ACME: An Architecture Description Interchange Language. In: Proceedings of CASCON'97. IBM Press, Cranbury, NJ, USA. pp 169–183

Garlan D, Schmerl B, Cheng SW (2009) Software Architecture-Based Self-Adaptation. In: Zhang Y, Yang LT, Denko MK (eds). Autonomic Computing and Networking. Springer, US. pp 31–55. Chap. 2

Garlan D, Cheng SW, Huang AC, Schmerl B, Steenkiste P (2004) Rainbow: Architecture-Based Self-Adaptation with reusable infrastructure. Computer 37(10):46–54

Gat E (1998) Three-layer architectures. In: Kortenkamp D, Bonasso RP, Murphy R (eds). Artificial Intelligence and Mobile Robots. MIT Press, Cambridge, MA, USA. pp 195–210

Grassi V, Mirandola R, Randazzo E (2009) Model-Driven assessment of QoS-aware Self-Adaptation. In: Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science, vol. 5525. Springer, Berlin, Heidelberg. pp 201–222

Grassi V, Mirandola R, Sabetta A (2007) A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In: Proceedings of WOSP '07. ACM, New York, NY, USA. pp 103–114

Guck D, Han T, Katoen JP, Neuhäußer MR (2012) Quantitative timed analysis of interactive markov chains. In: Goodloe AE, Person S (eds). NASA Formal Methods. Lecture Notes in Computer Science, vol. 7226. Springer, Berlin, Heidelberg. pp 8–23

Hallsteinsen S, Geihs K, Paspallis N, Eliassen F, Horn G, Lorenzo J, Mamelli A, Papadopoulos GA (2012) A development framework and methodology for self-adapting applications in ubiquitous computing environments. J Syst Softw 85(12):2840–2859

Hermanns H (2002) Interactive Markov Chains: The Quest for Quantified Quality. Lecture Notes in Computer Science, Vol. 2428. Springer, Berlin, Heidelberg

Hermanns H, Katoen JP (2010) The how and why of interactive markov chains. In: Proceedings of FMCO'09. Lecture Notes in Computer Science, vol. 6286. Springer, Berlin, Heidelberg. pp 311–337

Hnětynka P, Plášil F (2006) Dynamic reconfiguration and access to services in hierarchical component models Component-Based software engineering. In: Component-Based Software Engineering. Lecture Notes in Computer Science, vol. 4063. Springer, Berlin, Heidelberg. pp 352–359. Chap. 27

CooPLa Team, CooPLa Editor (2014). http://coopla.di.uminho.pt

Huber N, Hoorn A, Koziolek A, Brosig F, Kounev S (2014) Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. Serv Oriented Comput Appl 8(1):73–89

Huebscher MC, McCann JA (2008) A survey of autonomic computing—degrees, models, and applications. ACM Comput Surv 40(3):1–28

IBM Corp (2004) An Architectural Blueprint for Autonomic Computing. IBM Corp, USA

Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50

Koehler C, Arbab F, Vink E (2009). In: Corradini A, Montanari U (eds). Reconfiguring Distributed Reo Connectors. Lecture Notes in Computer Science, vol 5486. Springer, Berlin, Heidelberg. pp 221–235

Kramer J, Magee J (1990) The evolving philosophers problem: Dynamic change management. IEEE Trans Softw Eng 16(11):1293–1306

Krause C (2011) Reconfigurable component connectors. PhD thesis, Leiden University, Amsterdam, The Netherlands

Kwiatkowska M, Norman G, Parker D (2010) A framework for verification of software with time and probabilities. In: Proceedings of FORMATS'10. Lecture Notes in Computer Science, vol. 6246. Springer, Berlin, Heidelberg. pp 25–45
Litoiu M, Mihaescu M, Ionescu D, Solomon B (2008) Scalable adaptive web services. In: Proceedings of SDSOA '08. ACM, New York, NY, USA. pp 47–52
Losavio F, Chirinos L, Lévy N, Ramdane-Cherif A (2003) Quality characteristics for software architecture. J Object Technol 2(2):133–150
Moon Y, Arbab F, Silva A, Stam A, Verhoef C (2011) Stochastic Reo: a case study. In: Proceedings of the 5th International Workshop on Harnessing Theories for Tool Support in Software (TTSS '11), Oslo, Norway. pp 1–16
Moon YJ (2011) Stochastic models for quality of service of component connectors. PhD thesis, Universiteit Leiden
Moon YJ, Silva A, Krause C, Arbab F (2014) A compositional model to reason about end-to-end QoS in stochastic Reo connectors. Sci Comput Programm 80:3–24
Nallur V, Bahsoon R (2013) A decentralized self-adaptation mechanism for service-based applications in the cloud. Softw Eng IEEE Trans 39(5):591–612
Nilsson NJ (1980) Principles of Artificial Intelligence. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
Oliveira N, Barbosa LS (2013a) On the reconfiguration of software connectors. In: Proceedings of SAC'2013, vol 2. ACM, New York, NY, USA. pp 1885–1892
Oliveira, N, Barbosa LS (2013b) Reconfiguration mechanisms for service coordination. In: her Beek MH, Lohmann N (eds). Web Services and Formal Methods. Lecture Notes in Computer Science, vol. 7843. Springer, Berlin, Heidelberg. pp 134–149
Oliveira N, Barbosa LS (2014) A self-adaptation strategy for service-based architectures. In: VIII Brazilian Symposium on Software Components, Architectures and Reuse. SBCARS'2014, vol. 2. SBC - Brazilian Computer Society, Porto Alegre, RS, Brazil. pp 44–53
Oliveira N, Silva A, Barbosa LS (2014) Quantitative analysis of Reo-based service coordination. In: Proceedings of SAC'14. ACM, New York, NY, USA Vol. 2. pp 1247–1254
Oliveira, N, Silva A, Barbosa LS (2015) IMC_{Reo}: interactive Markov chains for stochastic Reo. J Internet Serv Inform Secur 5(1):3–28
Oreizy P, Gorlick MM, Taylor RN, Heimhigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL (1999) An architecture-based approach to self-adaptive software. Intell Syst Appl 14(3):54–62
Rodrigues F, Oliveira N, Barbosa LS (2014) 3rd Symposium on Languages, Applications and Technologies. OpenAccess Series in Informatics (OASIcs), vol 38. In: Pereira MJV, Leal JP, Simões A (eds). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. pp 61–76
Salehie M, Tahvildari L (2009) Self-adaptive software: Landscape and research challenges. ACM Trans Auton Adapt Syst 4(2):1–42
van der Aalst WMP, ter Hofstede AHM (2005) YAWL: yet another workflow language. Inform Syst 30(4):245–275
Villegas Machado NM, Müller HA, Tamura Morimitsu G (2011) On designing Self-Adaptive software systems. Sistemas & Telemática 9(18):29–51
Vromant P, Weyns D, Malek S, Andersson J (2011) On interacting control loops in self-adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS '11. ACM, New York, NY, USA. pp 202–207
Wermelinger MA (1999) Specification of software architecture reconfiguration. PhD thesis, Universidade Nova de Lisboa, Lisboa, Portugal
Weyns D, Schmerl B, Grassi V, Malek S, Mirandola R, Prehofer C, Wuttke J, Andersson J, Giese H, Göschka K (2013) On patterns for decentralized control in Self-Adaptive systems. In: de Lemos R, Giese H, Müller H, Shaw M (eds). Software Engineering for Self-Adaptive Systems II. Lecture Notes in Computer Science, vol. 7475. Springer, Berlin Heidelberg. pp 76–107