Performance Aware Open-world Software in a 3-Layer Architecture *

Diego Perez-Palacin
Dpt. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
Zaragoza, Spain
diegop@unizar.es

José Merseguer
Dpt. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
Zaragoza, Spain
jmerse@unizar.es

Simona Bernardi
Dip. di Informatica
Università di Torino
Torino, Italy
bernardi@di.unito.it

ABSTRACT

Open-world software is a new paradigm that stresses the concept of software service as a pillar for building applications. Services are unceasingly deployed elsewhere in the open-world and are used *on demand*. Consequently, the performance of these open-world applications relies on the performance of definitely unknown third-parties. Another consequence is that performance prediction methods can no longer assume that service times for software activities are well-known all over the time. More feasible solutions defend that they should be inferred from the environment, for example monitoring current services executions. So, there is a need for new performance prediction methods, and it is likely that they have to be applied not only when developing, but also during software execution, so to learn from the environment and to adapt to it. In this paper, we build on a three layer architecture, taken from literature, to present an architectural approach for performance prediction in open-world software. Once the approach is presented, the paper focuses on the intricacies of its more challeging component, i.e., the generator of strategies to meet performance goals by selecting the best available set of services.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*

General Terms

Design, Performance

Keywords

UML-MARTE, software components, self-managed systems, open-world software, Petri nets

*This work has been supported by the project DPI2006-15390 of the Spanish Ministry of Science and Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WOSP/SIPEW'10, January 28–30, 2010, San Jose, California, USA.
Copyright 2009 ACM 978-1-60558-563-5/10/01 ...$10.00.

1. INTRODUCTION

The open-world software paradigm [2] encompasses and abstracts concepts underlying a wide-range of approaches and technologies; among them, grid computing, publish-subscribe middleware or service oriented architectures. In open-world, an accepted approach considers software as made of *services* provided by *components* elsewhere deployed that interplay without authorities. The software achieves its goals by selecting and adapting services which evolve independently. Then, this software evolves itself in unforeseen manners that depend on third-parties, which means that the performance for this software strongly relies on that of the services it trusts. Therefore, the methods in the software performance field proposed so far to predict "non open-software" can now hardly be completely reused in this new context. Consider they make assumptions which now could not take place, for example, to assume durations of software activities as well-known performance input parameters.

Kramer and Magee in [7, 8] proposed an architecture for self-managed systems, i.e., those which are capable of self-configuration, self-monitoring and self-tuning. When a self-managed system suffers from dynamic changes during operation, it should configure itself to satisfy the specification or it may be capable of reporting that it cannot. Kramer and Magee defend that an architectural approach for this kind of systems brings several benefits; among others, generality to be applied in different domains, abstraction in the composition, scalability or potential for an integrated software approach.

Being self-management an inherent characteristic in open-world software, it is argued that challenges in the former are also present in the latter. Hence, we are convinced that open-world can take advantage of the Kramer and Magee three-layer reference architecture (KM-3L). At this respect, we want to study if KM-3L fits the open-world and if it can bring those previously enumerated benefits to this context. In particular, we will focus this work on how to exploit KM-3L for the open-world software to incorporate a performance-aware property. In fact, we quest for a reference architecture that allows systems to configure themselves, during execution, for satisfying some performance goal, as long as functionality is preserved. In this regard, the contributions of the paper are:

- First, we discuss how open-world software could be adapted to KM-3L. In particular, we stress the implications for KM-3L to carry out performance-aware reconfigurations in this context. We will accomplish it in Section 3.

- Once the architectural implications for performance has been presented, we will address an explanation about the most challenging component in this architecture. This is the com-

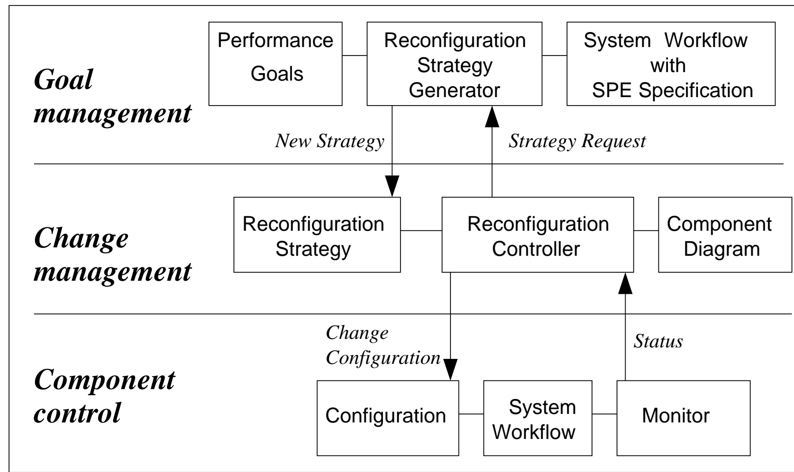

Figure 1: 3-layer architecture adapted to open-world

ponent in charge of generating the *strategies* that know how to carry out performance-aware reconfigurations. Section 4 describes algorithms for this component.

- The last contribution is an example, developed in Section 5, that demonstrates the feasibility of the proposed module and shows how the strategies it develops may improve the system performance.

The paper will end up in Section 6 with a brief conclusion, a discussion of the closest related work and ideas about future work. Next section summarizes this Kramer and Magee's vision of a reference architecture for self-managed systems (KM-3L).

2. A 3-LAYER ARCHITECTURE

The KM-3L proposal was inspired by the architectures developed for robotics systems, in particular following Gat's description in [4]. Indeed, as commented by Kramer and Magee, both robotics and self-managed systems are kinds of autonomic systems. For them, the idea is not to propose an implementation architecture but to identify what a self-managed system needs to carry out its mission, of course without human intervention. In the following we describe KM-3L by referencing for each layer its goals.

Component control.

This layer is made of those components that make up the self-managed system. It senses and reports the context *status* to its upper layer.

Change management.

This layer has a set of plans or *strategies* to achieve the system goal or *mission*. When the lower layer reports here the current context *status*, it can mean for this layer to produce a new configuration. For this purpose, it executes the strategies to change the underlying component architecture into one that fits with the current context or environment. This may imply either to introduce new components or to change the interconnections or the component parameters. When the environment *status* reported is not supported by any of the existing strategies, then this layer asks the upper one for a new strategy to manage the situation.

Goal management.

This layer manages the system mission and has to produce strategies that satisfy the mission taking into account the current environment. The strategies are produced when the mission changes as well as when the change management layer requests.

Once summarized the layers which compose this architecture, we just remark that the duration of the activities has been the criteria taken into account for placing a function in a given layer. Hence, immediate activities appear in lower levels, so for the system to quickly react to changes in the environment. However, long term activities are accomplished by the upper layer that may involve deliberation.

3. 3-LAYER ARCHITECTURE FOR OPEN-WORLD SOFTWARE

In this section, we describe how to adapt KM-3L to the open-world software context, and at the same time how to manage the performance-aware property. Then, for each layer we have to identify what responsibilities it has to take so the system eventually can accomplish this property. Hence, we are pursuing an architecture for performance-aware open-world software.

Concerning the architecture, we keep the point in the previous section, therefore we want it to be a reference, then we aim at identifying these responsibilities and their purpose in the overall of this comprehesive goal.

3.1 Component Control

As previously discussed, this layer is in contact with the execution environment and has to quickly react to changes produced in it. In the open-world software this means that this layer manages the components making up the current configuration. Therefore, it is responsible for establishing the current bindings and unbindings when a component has to be called.

Concerning performance, we identify for this layer different responsibilities. They are the minimum set an open-world software may need to actually develop activities leading to manage performance aware reconfigurations. Firstly, it will be in charge of tracking the performance of the services involved in the current configuration. Secondly, it has to discover new components offering services equivalent in functionality to those required by the workflow. Finally, it has to be aware about which ones of the current providers are no longer available.

For an open-world software to carry out these responsibilities it would be of interest to construct a *monitor* module that takes charge of all them. This monitor should be incorporated to the target open-world software as a module. For the first task, it will control the time elapsed in the calls to the services and for the second and third it will use the normal means in open-world (i.e., through service discovering).

From a practical point of view, this layer also needs a representation of the *workflow* to be executed and of the set of components that conform the current *configuration*. In this work, we will consider that such workflow has the form of a UML activity diagram while the current configuration will be represented by a UML component diagram (indeed an instance of the one in the Change Management level). Whatever other standard representation could be valid such as BPEL for the first or Darwin component model for the latter.

This layer reports the current *status* to the upper one each time the execution of a service ends (to inform about the monitored time), but also when it cannot execute the current service in the workflow (e.g., the target component may be unreachable). The upper layer can respond with a new configuration.

3.2 Change Management

The *mission* of an open-world software is obviously carried out through its own execution, here abstracted by the workflow. The workflow execution may need successive self-reconfigurations, that may attend different criteria, for example the cost of the services or the performance. For each criteria of interest, this level can associate at least one *reconfiguration strategy*. It would also be desireable that a given strategy could gather more than one criteria, for example the previous two. In any case, for this paper purposes the interest is that this layer has defined and can manage a performance aware reconfiguration strategy.

For an open-world software to execute strategies, we identify the need of a *reconfiguration controller* module. The inputs for this module would be of course the set of strategies, but also, the *status* provided by the monitor and a UML component diagram (CD). The output will account for the computed new system configuration. The CD describes for each component its mode, later explained. The status is the subset of currently active components in the CD.

Let us briefly discuss how this layer could manage the components *mode*. The *mode* can be a tuple `<state,MST>`. The first field to be chosen from `{unavailable,standby,active}` and the second to represent the mean service time for the module. Following the proposal in [7], the mode could be managed through ports using a `setmode` operation.

Moreover, this layer should `create` the new components and `delete` those no longer useful, remember that the actual `bind` and `unbind` is responsibility of the lower level. Therefore, when the monitor reports the status, this layer has to manage different situations:

- *A component is no longer available*. The reconfiguration controller (ReCtrl) sets the mode to *unavailable*, and if the component is in use then the ReCtrl executes the strategy to find a proper substitute and eventually will report a configuration change.
- *A provider is available for a given service*. The *status* here reported has to include provider's and service's name and the service MST `t`. As long as this provider has a CD entry, the reconfiguration controller updates it with the new service as `<standby,t>`. Otherwise, it performs a `create` for the provider (as a component) and sets service mode as `<standby,t>`.
- *A service is currently not providing the required QoS*. The reconfiguration controller executes the strategy and decides about a service change. If the change is necessary, then it sets the mode of the degraded component from <`active`,t_1> to <`standby`,t'_1> and the mode of the one selected from <`standby`,t_2> to <`active`,t_2>. When the new configuration will be reported, the lower lever takes the responsibility to perform the corresponding `unbind` and `bind`.

Sometimes the current strategy cannot produce a new configuration for the reported *status* (e.g., the selected providers are not available or the performance goal cannot be satisfied). Then this layer will request the *Goal Management* for a new performance aware strategy.

Finally, we remark that the operations in this layer (`create, delete, setmode` and the strategy execution) are supposed to be immediate regarding the system execution time. This is important since this level will not overload the system.

3.3 Goal Management

From our point of view, the *mission* of the system will be not only to carry out the workflow functionality, but also to do it meeting a *performance goal*. For us this layer has to produce performance aware reconfiguration strategies, then we devise a *strategy generator* module. Irrespective of this module, the system could create strategies to meet other goals of interest in the scope of the open-world software.

The strategies are provided on the *Change Management* layer demand and they could be afforded under two assumptions:

- There could exist a library of strategies and the *generator* will decide the appropriate one, for the current request, out of this set.
- The *generator* could actually create the strategy on demand.

In this work we just explore the second choice, then the *generator* inputs should be: the *performance goal*, the workflow with a specification of certain performance properties, and the current configuration that will be provided with the *Change Management* request. The output is the target strategy that meets the defined *performance goal*, for the sake of simplicity we will consider only system response time. The performance specification will use the MARTE [12] profile.

4. GENERATION OF STRATEGIES

In this section, we offer a high-level view of the reconfiguration *strategy generator* module, which is placed in the *Goal Management* layer. The previous section described the module goal and its interfaces. Algorithms 1, 2 and 3 synthesize the module functionality, i.e., they report to the *Change Management* layer the strategy they obtain. Besides, a warning complements the strategy when it does not meet the performance goal.

Information managed in the algorithms.

We assume that the system workflow needs to call K external services, s_k, $k \in [1..K]$. In the CD in Fig. 3, $K = 3$, thought that the same service could be requested in different calls.

A given service s_k may be provided by several components; let L_k be the number of components that provide s_k. We denote as c_{kl}, where $k \in [1..K]$ and $l \in [1..L_k]$, the lth component serving s_k. For example, in Fig. 3, service s_3 is served by two components c_{31}, c_{32}. Then, $C_k = \bigcup_{l=1}^{L_k} c_{kl}$ is the set of all components that offer

s_k, and $C = \bigcup_{k=1}^{K} C_k$ is the set of all components that provide the services specified in the system workflow.

Moreover, we assume that there exists a Time Table (TT), like Table 1, describing for each component its working *phases*. Let J_{kl} be the number of working phases of component c_{kl}; then each phase ph_j, where $j \in [1..J_{kl}]$, is characterized by a pair of real values (S_j^{kl}, SJ_j^{kl}), where S_j^{kl} is the mean service time and SJ_j^{kl} is the mean sojourn time of c_{kl} in ph_j. This timing information would come from the providers or from our experience monitoring the environment.

A reconfiguration strategy is represented as a directed graph $G = (N, E)$, see example in Fig. 5. A node $n \in N$ is interpreted as a system configuration, but it is also important to know for each component the *phase* we guess it is working out. An edge $e \in E$ is interpreted either as a change of system configuration, i.e., the component c_{kl_1} that offers service s_k will replace the component c_{kl_2} ($l_1 \neq l_2$) that provides the same service, or as a change in a component phase. For example, in Fig. 5, the egde from $Node_0$ to $Node_2$ represents a change of system configuration (c_{22} will replace c_{21}), while the edge from $Node_0$ to $Node_1$ models a change of phase in component c_{11} (from ph_1 to ph_2).

An edge is labeled as $\langle s_k, cond \rangle$, where s_k is the service and $cond$ is a ratio representing our minimun *confidence level* for the change to be produced, consider that being stochastic our analyses, then there exist a probability that the strategy fails its prediction.

Description of the algorithms.

Algorithm 1 summarizes the strategy generation, it starts creating the strategy initial node (line 2) and from this node produces its adjacent ones (line 11) and the edges that join them (line 16). While there are nodes whose outgoing edges have not been created yet, it keeps creating nodes and edges. Finally, it creates a set of "way back" edges (line 21). Such edges represent either changes of configuration or component-phases due to timeouts instead of a *condition* as it happened to forward edges. The rational behind a "way back" is to bring back the system to a configuration that after some time would be working better than the current one.

Algorithm 2 solves the calls in Algorithm 1 (lines 2,11), i.e. how to create a node in the strategy. *PhaseList* is a list of pairs $\langle c_{kl}, ph_j \rangle$ that for each $c_{kl} \in C$ assumes its phase $ph_j \equiv (S_j^{kl}, SJ_j^{kl})$. When Algorithm 2 creates the initial node, *PhaseList* (line 3) is created assuming that each c_{kl} is in its phase with minimum mean service time. However, for the rest of the nodes (line 5), *PhaseList* is constructed with *ExtractListOfPhases* that will implement an algorithm choosing appropriate phases. In Section 5 we will exemplify our proposal for such algorithm. Function *AllPossibleConfigs* (line 9) creates all possible system configurations according to *PhaseList*. Each configuration will parameterize the workflow GSPN that will be evaluated to get the configuration response time (lines 11..13). In particular, the mean service times S_j^{kl} of the components c_{kl} belonging to the configuration, in their current phase ph_j, are used to parameterize the GSPN. As an example, during the creation of the initial node $Node_0$ in Fig. 5, four candidate configurations are generated (see Table 2) and evaluated using the workflow GSPN in Fig. 4. Finally, the node created by the Algorithm 2 (line 15) corresponds to the system configuration with the minimum response time.

Algorithm 3 solves the call in Algorithm 1 (line 16), i.e. how to create a forwarded edge, not a "way back". Observe that, if service s_k of a given $Node_s$ cannot be replaced by any $Node_t$ with a new *phase* or component, no edge is created (line 1). Function *ExtractListOfPhases*, in Algorithm 2, detected this situation and *CreateNode* returned null. When $Node_s$ has an adjacent node $Node_t$, then a direct edge from the source node $Node_s$ to the target node $Node_t$ is created together with its labeling information, i.e., the service s_k and the condition $cond$ (line 5). In particular, $cond$ is a real value computed by the function *SetConfLevel* (line 10), which needs the response time evaluated using the workflow GSPN for $Node_s$ and $Node_t$ (lines 6-9). A simple example of *SetConfLevel* will be given in Section 5.

Algorithm 1 Strategy generation

Require: From Goal Management Layer: System Workflow (AD), Performance Goal (PerfGoal)
From Change Management Layer: Components with their timing specification (CD,TT)
Ensure: A New Strategy (and a possible warning meaning that the PerfGoal is not achieved)
{*Initialization*}
1: **set** $G = \langle N, E \rangle$: $N = \emptyset$ {nodes}, $E = \emptyset$ {edges}
{*Create Initial Node*}
2: $N_0 \leftarrow$ CreateNode(AD,CD,TT,null,null)
3: **set** $Nodes = \emptyset$
4: $Nodes = Nodes \cup N_0$
5: **while** $Nodes \neq \emptyset$ **do**
6: $Node_s \leftarrow$ ExtractOneNode($Nodes$)
7: AlreadyCreated $\leftarrow$ CheckNode(N, $Node_s$)
8: **if not** AlreadyCreated **then**
9: $N \leftarrow N \cup Node_s$
{*Create $Node_s$ adjacent nodes*}
10: **for all** $k \in [1..K]$ **do**
11: $Node_t \leftarrow$ CreateNode(AD,CD,TT,k,$Node_s$)
12: AlreadyCreated $\leftarrow$ CheckNode(N, $Node_t$)
13: **if not** AlreadyCreated **then**
14: $Nodes \leftarrow Nodes \cup Node_t$
15: **end if**
{*Create edge from $Node_s$ to $Node_t$*}
16: $Edge \leftarrow$ CreateEdge($Node_s$,$Node_t$,k,TT)
17: $E \leftarrow E \cup Edge$
18: **end for**
19: **end if**
20: **end while**
21: $E = E \cup$ CreateWayBackEdges(G,TT)
22: **return** <G, AnalyseStrategy(G,PerfGoal,CD,TT)>

5. EXAMPLE

We exemplify the algorithm of the strategy generation, described in Section 4, with an example of a system under development (SUD) that executes three operations, in a sequential manner. All such operations consist in service calls to providers in the open-world environment. The UML system specification is shown in Figures 2 and 3. The activity diagram (Figure 2), annotated with the MARTE profile [12], represents the system workflow. The type of workload (*GaWorkloadEvent*) is open and requests arrive to the SUD with an exponential inter-arrival time, with a mean of 500 time units (i.e., "tu"). The requests are processed, one at a time, by acquiring (*GaAcqStep*) and releasing (*GaRelStep*) the resource c_0. Each activity step (*PaStep*) models an external service call s_k to a provider in the open-world. In particular, the *extOpDemands* tagged-value is a parameter that is set to the current provider of service s_k and the *extOpCount* tagged-value indicates the number of requests made for each service call.

The component diagram (Figure 3) represents the currently available providers of the services required by the system. In partic-

Algorithm 2 CreateNode

Require: AD,CD,TT,service (k), current node (node)
Ensure: A node ($\mathrm{conf_{best}}$)
1: **set** PhaseList = $\emptyset$ {vector of vectors}
2: **if** (k==null $\wedge$ node==null) **then**
3: PhaseList $\leftarrow$ ExtractInitialListOfPhases(CD,TT)
4: **else**
5: PhaseList $\leftarrow$ ExtractListOfPhases(CD,TT,node,k)
6: **end if**
7: **set** CandidateConfigs = $\emptyset$ {set of configurations}
8: **set** RTs = $\emptyset$ {set of configuration response times}
9: CandidateConfigs $\leftarrow$ AllPossibleConfigs(PhaseList)
10: **for all** conf $\in$ CandidateConfigs **do**
11: $\mathrm{GSPN_{conf}}$ $\leftarrow$ CreateGSPN(conf)
12: $\mathrm{rt_{conf}}$ $\leftarrow$ Evaluate($\mathrm{GSPN_{conf}}$)
13: RTs $\leftarrow$ RTs $\cup$ $\langle \mathrm{conf}, \mathrm{rt_{conf}} \rangle$
14: **end for**
15: $\mathrm{conf_{best}}$ $\leftarrow$ FindBestConfig(RTs)
{The node is a configuration with the min response time: $\langle \mathrm{conf_{best}}, \mathrm{rt} \rangle \in$ RTs | $\forall \langle \mathrm{conf}, \mathrm{rt_{conf}} \rangle \in$ RTs : $\mathrm{rt} \leq \mathrm{rt_{conf}}$ }
16: **return** $\mathrm{conf_{best}}$

Algorithm 3 CreateEdge

Require: source ($Node_s$), target ($Node_t$), service (k), TT
Ensure: The edge between $Node_s$ a $Node_t$ (edge)
1: **if** $Node_t$ == **null** **then**
2: **return null**
3: **end if**
4: **set** cond = 0.0 {confidence level (float)}
5: **set** edge = $\langle Node_s, Node_t, k, \mathrm{cond} \rangle$
{Computation of $Node_s$ response time}
6: GSPN_{Node_s} $\leftarrow$ CreateGSPN($Node_s$)
7: rt_{Node_s} $\leftarrow$ Evaluate(GSPN_{Node_s})
{Computation $Node_t$ response time}
8: GSPN_{Node_t} $\leftarrow$ CreateGSPN($Node_t$)
9: rt_{Node_t} $\leftarrow$ Evaluate(GSPN_{Node_t})
{Computation of the confidence level}
10: cond $\leftarrow$ SetConfLevel($Node_s, rt_{Node_s}, Node_t, rt_{Node_t}$,TT)
11: **return** edge

ular, component's names are given according to the name of the service they provide. There exists only one provider c_{11} of service s_1, while two providers are available for each service s_2 and s_3. Table 1 (TT) shows the working phases, in time units, of the providers. In particular, for each provider c_{kl}, the estimated mean service times S_j^{kl} and mean sojourn times SJ_j^{kl} of the offered service, are given.

5.1 Strategy Generation

The Time Table and the UML specification, properly annotated with MARTE, provide the input for the Algorithm 1 described in Section 4. A parametric GSPN model is then created from the activity diagram (Figure 2) that will be used to estimate the mean response time of the system under different configurations, using the `multisolve` facility of GreatSPN [6]. The GSPN model is shown in Figure 4 and it is characterized by three rate parameters representing the execution mean rates of the service calls s_1, s_2 and s_3.

Observe that the call to service s_2, in the activity diagram, includes 3 requests (*extOpCount* tagged-value) this is modeled by the free-choice subnet, where the weights assigned to the conflicting

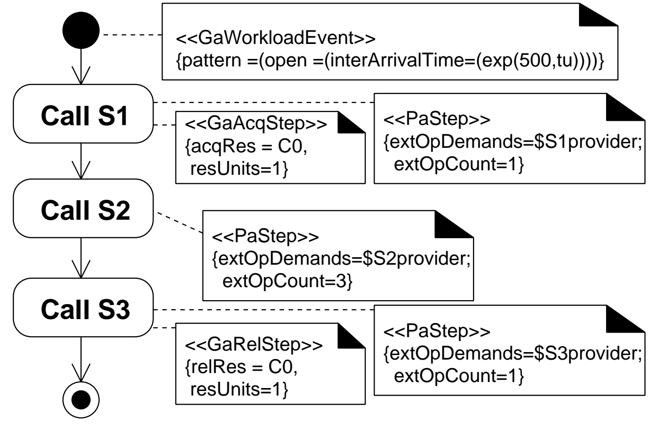

Figure 2: UML activity diagram

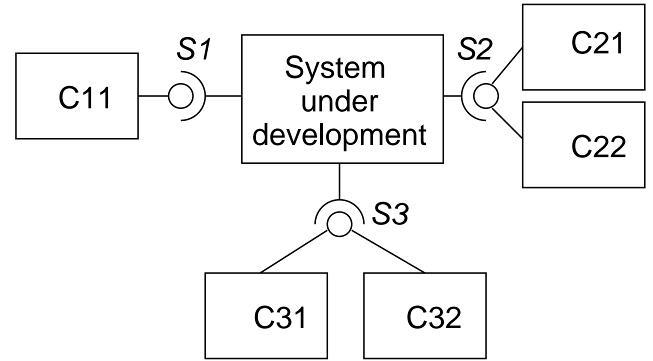

Figure 3: UML component diagram

transitions *Start_CallS2* and *End_CallS2* are equal, respectively, to $3/4$ and $1/4$.

The first main step of the algorithm (Algorithm 1 - line 2), consists of creating the initial node of the reconfiguration strategy graph (Algorithm 2). This is accomplished by assuming that each provider works under the best mode. We consider, then, the minimum estimated (mean) service times from each provider, i.e., $S_1^{11} = 5tu$, $S_1^{21} = 10tu$, $S_1^{22} = 35tu$, $S_1^{31} = 20tu$ and $S_1^{32} = 30tu$. There are four possible system configurations: for each one, we instantiate the parametric GSPN, in Figure 4, by setting the rate parameters $\lambda_{S1provider}$, $\lambda_{S2provider}$ and $\lambda_{S3provider}$ to the inverse of the considered service times $S_1^{kl} (k = 1, 2, 3)$ of each current provider of services s_1, s_2 and s_3, respectively. Once instantiated, the GSPNs are solved and the system (mean) response times are computed (see Table 2).

In the strategy graph (Fig. 5), the initial node $Node_0$ corresponds to the configuration that revealed the minimum system (mean) response time. Observe that, in this simple example, active providers in the initial configuration correspond to those ones having the minimum service times. However, this property does not always hold in a general case where several providers contend for shared resources.

In the next main step of the Algorithm 1 (line 11), the nodes adjacent to the initial one are created, considering that the active providers in $Node_0$ can degrade their performance. Eventually, there will be three configuration nodes adjacent to the initial node, one for each external service requested by the SUD (Figure 5). Let us consider the creation of the first two nodes $Node_1$ and $Node_2$

Provider working phases (in time units, i.e., tu)			
	ph_1	ph_2	ph_3
C11	(5,3000)	(20,6000)	
C21	(10,6000)	(70, 2000)	(250,2000)
C22	(35,6000)	(140,4000)	
C31	(20,2000)	(70,2000)	
C32	(30,∞)		

In format $ph_j = (S_j^{kl}, SJ_j^{kl})$

Table 1: Time Table of open-world providers (TT)

Mean response time estimation (in time units, i.e., tu)	
C11:ph_1 C21:ph_1 C31:ph_1	60.5
C11:ph_1 C22:ph_1 C31:ph_1	177.6
C11:ph_1 C21:ph_1 C32:ph_1	72.5
C11:ph_1 C22:ph_1 C32:ph_1	193.8

Table 2: System components candidates

adjacent to $Node_0$: the algorithm will iterates over the created nodes to produce their adjacents, until all the possible system configurations are examined.

$Node_1$ is added considering that the active provider of service s_1 in $Node_0$ (i.e., c_{11}) changes its *phase* from ph_1 to ph_2, i.e., c_{11} is answering to service requests with a mean service time of $20tu$, instead of $5tu$. Since c_{11} is the unique provider of s_1, the $Node_1$ is characterized by the same active providers as $Node_0$ as well as the same provider mean service times but the one of c_{11}, which is equal to $20tu$. The GSPN model in Figure 4 is used to compute the system mean response time of the configuration $Node_1$.

$Node_2$ is created assuming that the active provider of s_2 in $Node_0$ (i.e., c_{21}) changes its *phase* by increasing the mean service time from $10tu$ to $70tu$. Then, four candidate configurations were possible: two of them still include c_{21} as active provider of s_2 with degraded performance. They correspond to the first and the third configuration in Table 2 with the provider c_{21} in phase ph_2. In the other two configurations, the active provider of s_2 is c_{22} (i.e, the second and the fourth configuration in Table 2). The GSPN model in Figure 4 is then used to select the best configuration among the candidates, that is the one with the minimum system (mean) response time. Then, the $Node_2$ actually corresponds to the configuration with the minimum system (mean) response time, i.e., $177.6tu$.

Once a new adjacent node is created, the algorithm generates the corresponding forward edge (Algorithm 1- line 16). An edge from $Node_s$ to $Node_t$ includes information about the service s_k and the goodness of the prediction (confidence-level) for the *reconfiguration controller* to decide whether it is worth to change the configuration from $Node_s$ to $Node_t$. Observe that, since we are dealing with the open-world environment, every decision about the providers is based on predictions. We propose an ad-hoc heuristic that works under the open workload assumption and considers the performance goal (i.e., obtain the best system mean response time) as well as the available timing specifications (i.e., provider working phases).

Let us consider an edge from $Node_s$ to $Node_t$ where the source and the target nodes have different active components, such as $Node_0$ and $Node_2$ in Figure 5. The computation of the corresponding minimum confidence level is related to two quantities:

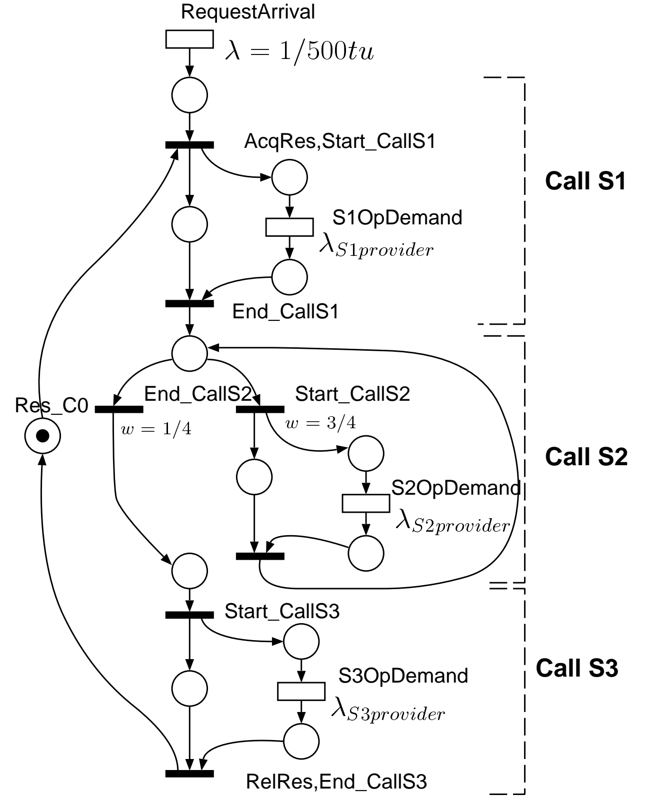

Figure 4: Parametric GSPN

- The performance improvement when the system reconfigures properly, that is the provider has changed its phase and the strategy realizes it (e.g., the provider c_{21} has changed from ph_1 to ph_2 and the system moves from $Node_0$ to $Node_2$). This is estimated as:

$$Perf_{improve} = rt_{s|c_{kl} \leftarrow ph_{j+1}} - rt_t,$$

where $rt_{s|c_{kl} \leftarrow ph_{j+1}}$ is the system mean response time with the same active providers as in $Node_s$, but changing the working phase of provider c_{kl} from ph_j to ph_{j+1}, and rt_t is the system mean response time in $Node_t$.

- The performance loss when the system reconfigures due to a wrong prediction, that is the provider has occasionally had a slow execution, but it has not really changed its current phase, however the system moves to the target node. This is estimated as:

$$Perf_{loss} = rt_t - rt_s,$$

where rt_s is the system mean response time in $Node_s$.

Then, the minimum confidence level is given by the formula:

$$conf_level = \frac{Perf_{improve}}{Perf_{improve} + Perf_{loss}}. \qquad (1)$$

When the source and target nodes of an edge have the same active components, such as $Node_0$ and $Node_1$, the minimum confidence level is computed as $conf_level = \frac{rt_s}{rt_t}$.

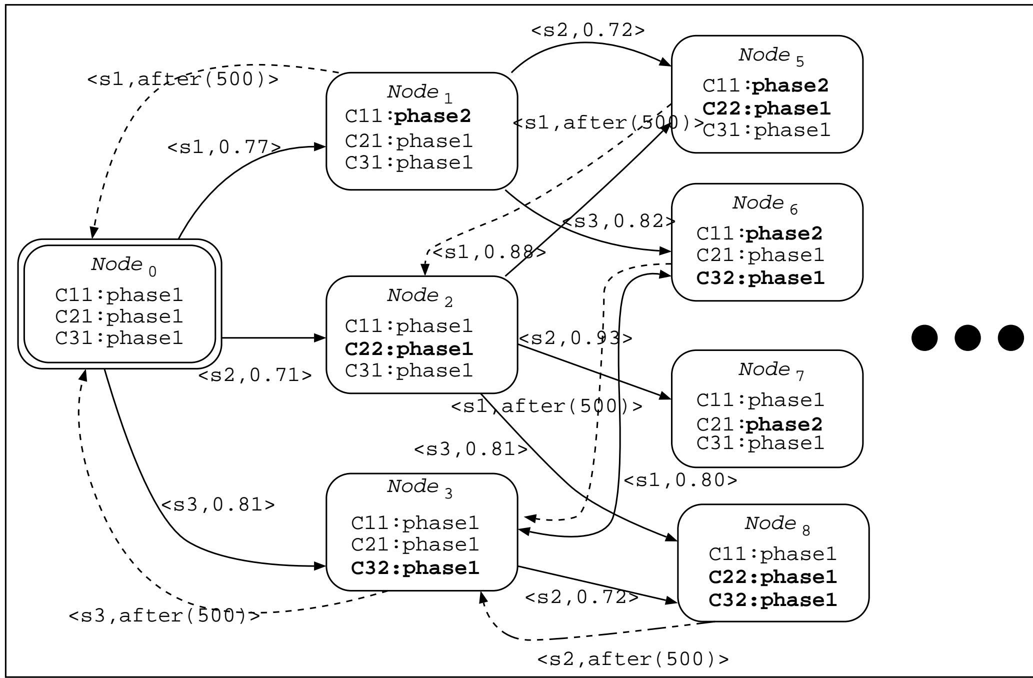

Figure 5: Partial reconfiguration strategy graph

Finally, the *way-back* edges are created (Algorithm 1 - line 21) to allow the system to move back to a previously considered configuration after a (mean) sojourn time period in the source node. So there will be an edge from $Node_s$ to $Node_t$, labeled with a mean sojourn time period as a timeout, if there exists a provider c_{kl} in $Node_s$ with its final phase $ph_{J_{kl}}$ and in $Node_t$ with its initial phase ph_1. In Figure 5, way-back egdes are dashed and, for readability, only five of them are shown. The choice of the ideal mean sojourn time period that allows the system to achieve the performance goal (i.e., minimum response time) is a future work issue. In the example, we set such period equal to the mean inter-arrival time of a service request to the SUD (i.e., $500tu$).

In order to validate our proposal, we carried out the analysis of the system, considering several assumptions: the system does not follow the strategy modeled by the reconfiguration graph in Figure 5 (case 1), and the system undergoes reconfigurations according to the strategy graph (case 2). We obtained the following results for the system mean response time: $494tu$ (case 1) and $436tu$ (case 2). This means that partially applying our performance aware reconfiguration (eight nodes in Fig. 5) we have improved the system response time in 11%.

6. CONCLUSION AND RELATED WORK

During this paper elaboration, we have learnt that there exist a lot of challenges for the performance prediction of the open-world software to become a reality. However, we believe that this paper has proposed a clear reference architecture, which means an attempt to comprehensively accomplish most of such challenges. From this architecture, we have explored how to generate strategies, that can reconfigure a system while its performance goal has to be achieved. Our *generation* technique tried to show up where the problems are and it demonstrates a possible solution using Petri nets. However other generation approaches could be feasible and would be desirable, we validated our solution through an example. The future work has to address all these open challenges to get a real comprehensive proposal. Besides performance, other properties such as dependability will be considered by our approach. As a technical detail, in this work we have not considered network transmission delays, however they can be easily incorporated through the UML deployment diagram.

Related work.

We believe that the idea of introducing a reference architecture coming from self-managed systems in the open-world software is original. Therefore, our solution to introduce and manage performance aspects in such architecture is also new. Probably, the closest work to ours is the one in [5], the authors also evaluate performance in open-world assuming components that can evolve independently and unpredictably. However, they use queueing networks and further comparisons are difficult since they address other challenges in the open-world instead of the strategy generation problem.

Although not focussed on the open-world paradigm, Menascé [15, 10, 11] evaluates service-based software. These works use brokers to negotiate and manage QoS parameters that are well-known and reliable. Our approach, that at this respect was inspired in [1, 9], is completely different since it tracks open-services to predict current QoS. This means that the quality of our predictions have to be of inferior quality, but consider that being open our environment, we have to deal with untrusted third-parties. Also in [14] is addressed the problem of guaranteeing the QoS of untrusted third-party services. They propose a framework to choose services offering best QoS, in this work the workload is balanced among several providers to support some kind of fault tolerance.

The work of Garlan in [3] also proposes an architecture for performance evaluation but restricted to self-healing systems, besides they do not use of formal methods. Oreizy et al. in [13] propose an architecture to manage the adaptation for evolvable systems, but this work does not deal with performance evaluation.

7. REFERENCES

[1] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *2nd International Conference on Service Oriented Computing*, pages 193–202, New York, NY, USA, 2004. ACM.

[2] L. Baresi, E. D. Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[3] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02*, pages 27–32, New York, NY, USA, 2002. ACM.

[4] E. Gat, R. P. Bonnasso, R. Murphy, and A. Press. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1997.

[5] C. Ghezzi and G. Tamburrelli. Predicting performance properties for open systems with kami. In *QoSA*, volume 5581 of *LNCS*, pages 70–85. Springer, 2009.

[6] The GreatSPN tool. http://www.di.unito.it/~greatspn.

[7] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[8] J. Kramer and J. Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, March 2009.

[9] R. Laddaga and P. Robertson. Self adaptive software: A position paper. In *SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems*, 2004.

[10] D. Menascé and V. Dubey. Utility-based QoS brokering in service oriented architectures. In *IEEE International Conference on Web Services*, pages 422–430, July 2007.

[11] D. Menascé, H. Ruan, and H. Gomaa. QoS management in service-oriented architectures. *Performance Evaluation*, 64(7-8):646–663, 2007.

[12] Object Management Group, http://www.promarte.org. *A UML Profile for MARTE.*, 2005.

[13] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[14] C. Patel, K. Supekar, and Y. Lee. A QoS oriented framework for adaptive management of web service based workflows. In *DEXA*, pages 826–835, 2003.

[15] M. Sopitkamol and D. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *WOSP'05*, pages 53–64. ACM, 2005.