Performance aware open-world software in a 3-layer architecture

Diego Pérez-Palacín
Dpto. de Informática e
Ingeniería de Sistemas
Universidad de Zaragoza
Zaragoza, Spain
diegop@unizar.es

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

Simona Bernardi
Dpto. de 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 the pillar to build applications. Services are continuously deployed elsewhere in the open-world and are used *on demand*. Consequently, the performance of these applications relies on the performance of probably unknown third-parties. Another consequence is that prediction methods can no longer assume that the service times for the software activities are well-known. More feasible solutions defend that they should be monitored. So, there is a need for new methods to predict performance and it is likely that they have to be applied also during software execution. 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.

Keywords

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

1. INTRODUCTION

The open-world software paradigm [1] encompasses and abstracts concepts underlying a wide-range of approaches and technologies; among them, grid computing, publish-subscribe middleware or service oriented architectures. In the 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 traditionally proposed in the software performance field to predict "non open-software" can now hardly be completely reused in this new context.

Kramer and Magee in [6, 7] 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. Obviously, open-world software also embraces the self-managed systems challenges. 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.

We are convinced that the open-world software paradigm can take advantage of the Kramer and Magee three-layer reference architecture (KM-3L). This should not only mean to integrate the benefits previously enumerated. In particular, we will study in this work how to exploit KM-3L for the open-world software to incorporate a performance-aware property, i.e. the system should configure itself to satisfy a performance goal. The contributions of the paper in this regard 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 component 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 the future work.

2. A 3-LAYER ARCHITECTURE

In this section we summarize the Kramer and Magee's vision of a reference architecture for self-managed systems (KM-3L). Their proposal was inspired by the architectures developed for robotics systems, in particular following Gat's description in [3]. 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.

The duration of the activities has been the criteria taken into account to place a function in a given layer. Hence, immediate activities appear in lower levels, so 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.

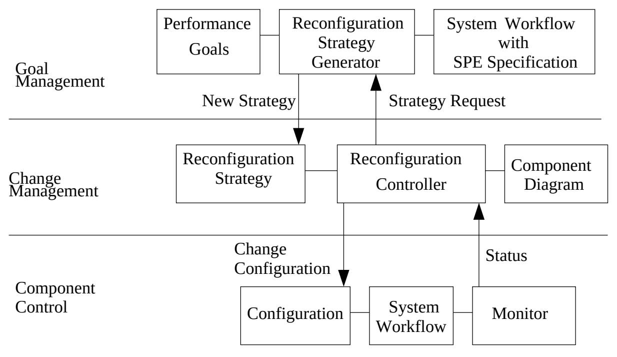

Figure 1: 3-layer architecture adapted to open-world

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 sucessive self-reconfigurations, which 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 [6], 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 the provider's and service's name and a MST `t` of the service. As long as this provider has an entry in the CD, 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 the 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 that 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 [8] profile.

4. GENERATION OF STRATEGIES

In this section, we offer a high-level view of the *generator* module. The previous section described the module goal and interfaces. Algorithms 1, 2 and 3 synthesize the module functionality, i.e. they report to the *Change Management* layer the strategy obtained. Besides, a warning is raised when the target strategy 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. Given service s_k, it may be provided by L^k components, so, $c_{kl} \mid k \in [1..K] \wedge l \in [1..L^k]$ is the l^{th} component serving s_k. In Fig. 3, service $s3$ is served by two components $c31, c32$, then $C^k = \bigcup_l c_{kl}$ is the set of all components that offer s_k, and $C = \bigcup_k C^k$

We assume that there exists a Time Table (TT), like Table 1, describing for each c_{kl} its working *phases*, a $phase_j$ is a pair $< mst_j, mjt_j >$, where $mst_j | j \in [1..J^{kl}]$ means the mean service time of c_{kl} in *phase* j, and mjt_j means its mean sojourn time. This information would come from the provider 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 is interpreted as a system configuration, but it is also important to know for each component the *phase* we guess it is performing. An edge is interpreted as a change of system configuration (i.e., c_{kl_1} will replace c_{kl_2} with $l_1 \neq l_2$; e.g. in Fig 5 from $Node_0$ to $Node_2$) or as a change in a component phase (e.g. in Fig 5 from $Node_0$ to $Node_1$). An edge is labeled as $< \; s_k, condition \; >$, where s_k is the service and $condition$ 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). It keeps creating nodes and edges, while there are nodes whose outgoing edges have not been created yet. Finally, it creates a set of "way back" edges (line 21), "ways back" represent 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 $< c_{kl}, phase >$ that for each $c_{kl} \in C$ assumes its phase. When Algorithm 2 creates the initial node, *PhaseList* (line 3) is created assuming that each c_{kl} is in its phase with minimum mst. However, for the rest of the nodes (line 5), *PhaseList* is constructed with *ExtractListOfPhases* that will implement an algorithm choosing appropiate phases. In section 5 we exemplify our algorithm proposal. Function *AllPossibleConfigs* (line 9) creates all the possible configurations with the nodes in the *PhaseList*. All these configurations will parameterize the workflow GSPN that will be evaluated to get its response time (lines 11..13). As an example, for $Node_0$ in Fig. 5, there will exist four possible configurations (see table 2). The mst_j in ph_j is used to actually parameterize the GSPN.

Algorithm 3 solves the call in Algorithm 1 (line 16), i.e. how to create a forwarded edge, not a "way back". If service s_k of a given $Node^s$ cannot be replaced by any $Node^t$ with a new *phase* or component then no edge is created (line 1). Function *ExtractListOfPhases* detected this situation and *CreateNode* will return null. When $Node^t$ exists, the edge is annotated with the source node, the target, and the labeling information, i.e. the s_k and the $condition$. Concerning the $condition$, it is created using function *SetConfLevel* (line 9) that needs the response time evaluated using the workflow GSPN for $Node^s$ and $Node^t$. 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 $G =< N, E >$: $N = \emptyset$ {nodes}, $E = \emptyset$ {edges}
    {Create Initial Node}
 2: $Node^0 \leftarrow$ CreateNode(AD,CD,TT,null,null)
 3: set $Nodes = \emptyset$
 4: $Nodes = Nodes \cup Node^0$
 5: while $Nodes \neq \emptyset$ do
 6:   $SourceNode \leftarrow$ ExtractOneNode($Nodes$)
 7:   AlreadyCreated $\leftarrow$ CheckNode($N, SourceNode$)
 8:   if not AlreadyCreated then
 9:     $N \leftarrow N \cup SourceNode$
        {Create SourceNode adjacent nodes}
10:     for all $k \in [1..K]$ do
11:       $TargetNode \leftarrow$ CreateNode(AD,CD,TT,k,$SourceNode$)
12:       AlreadyCreated $\leftarrow$ CheckNode($N, TargetNode$)
13:       if not AlreadyCreated then
14:         $Nodes \leftarrow Nodes \cup TargetNode$
15:       end if
16:       $Edge \leftarrow$ CreateEdge($SourceNode$,$TargetNode$,$k$,TT)
17:       E $\leftarrow$ E $\cup$ $Edge$
18:     end for
19:   end if
20: end while
21: $N = N \cup CreateWayBackEdges(G$,TT)
22: return <G, AnalizeStrategy(G,PerfGoal,CD,TT)>
```

```
Algorithm 2 CreateNode
Require: AD,CD,TT,service (k), previousNode (pnode)
Ensure: A node
 1: set PhaseList (Vector of vectors)
 2: if (k==null $\wedge$ pnode==null) then
 3:   PhaseList $\leftarrow extractInitialListOfPhases(CD, TT)$
 4: else
 5:   PhaseList $\leftarrow extractListOfPhases(CD, TT, pnode, k)$
 6: end if
 7: RTs = $\emptyset$ {set of response times}
 8: set PossibleConfigs = $\emptyset$
 9: PossibleConfigs $\leftarrow$ AllPossibleConfigs(PhaseList)
10: for all config $\in$ PossibleConfigs do
11:   $GSPN_{config} \leftarrow$ createGSPN(config)
12:   $rt_{config} \leftarrow$ evaluate($GSPN_{config}$)
13:   $RTs \leftarrow RTs \cup < config, rt_{config} >$
14: end for
15: $BestNode \leftarrow config \mid < config, rt_{node} > \in RTs \wedge$
    $\forall < rt_{config^i}, config^i > \in RTs, rt_{config} \leq rt_{config^i}$ {The
    resulting node is the best config based on response time}
16: return $BestNode$
```

5. CASE STUDY

```
Algorithm 3 CreateEdge
Require: Source Node ($Node^s$), Target Node ($Node^t$), service
    (k), TT
Ensure: Edge between $Node^s$ $Node^t$ with its information
 1: if $Node^t$ ==null then
 2:   return null
 3: end if
 4: set $edge =< Source, Target, service, condition >$:
    $Source = Node^s, Target = Node^t, service =$
    $k, condition = null$ {float, confidence level}
    {response time $Node^s$}
 5: $GSPN_{Node^s} \leftarrow$ createGSPN($Node^s$)
 6: $rt_{Node^s} \leftarrow$ evaluate($GSPN_{Node^s}$)
    {response time $Node^t$}
 7: $GSPN_{Node^t} \leftarrow$ createGSPN($Node^t$)
 8: $rt_{Node^t} \leftarrow$ evaluate($GSPN_{Node^t}$)
 9: $condition \leftarrow$ SetConfLevel($Node^s, rt_{Node^s}, Node^t, rt_{Node^t}$,TT)
10: return edge
```

We exemplify the algorithm of the strategy generation, described in Section 4, with a case study 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 [8], represents the system workflow. The type

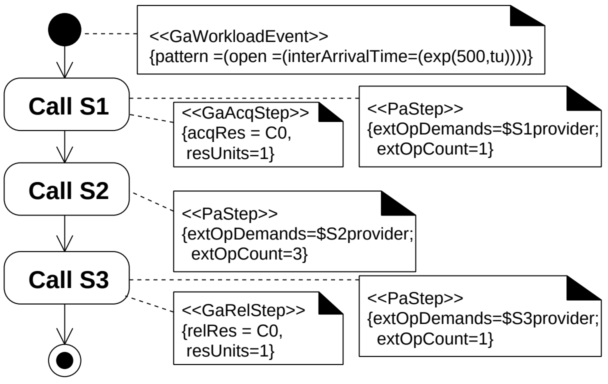

Figure 2: UML activity diagram

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 particular, 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, the estimated mean service times mst_i and mean sojourn time mjt_i of the offered service,

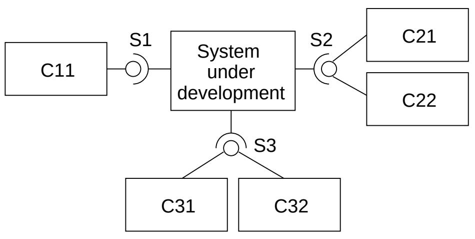

Figure 3: UML component diagram

are given.

Component service time estimation			
	$phase_1$	$phase_2$	$phase_3$
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 $phase_i = (mst_i, mjt_i)$

Table 1: Mean service times for open-world providers (TT)

5.1 Strategy Generation

The TT 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 [5]. 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 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., $c_{11}^{mst_1} = 5tu$, $c_{21}^{mst_1} = 10tu$, $c_{22}^{mst_1} = 35tu$, $c_{31}^{mst_1} = 20tu$ and $c_{32}^{mst_1} = 30tu$. 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 $c_{kl}^{mst_1} (k = 1, 2, 3)$ 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).

Mean response time estimation (p1=$phase_1$)			
C11:p1 C21:p1 C31:p1	60.5	C11:p1 C22:p1 C31:p1	177.6
C11:p1 C21:p1 C32:p1	72.5	C11:p1 C22:p1 C32:p1	193.8

Table 2: System candidates mean response time

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, the 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.

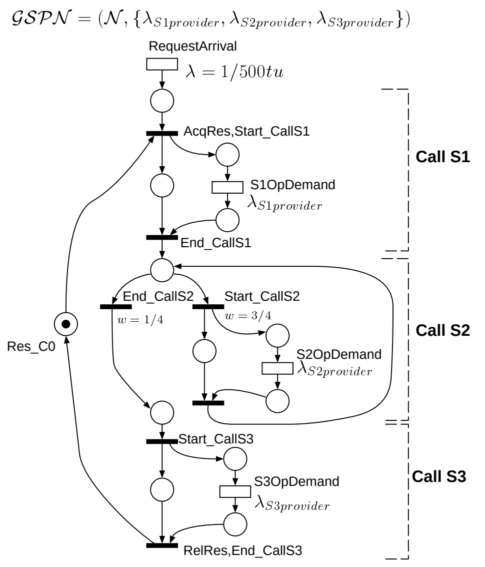

Figure 4: Parametric GSPN

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 $Node_2$ adjacent to $Node_0$: the algorithm will iterates over the created nodes to produce their adjacents, until all the possible system configurations are examined.

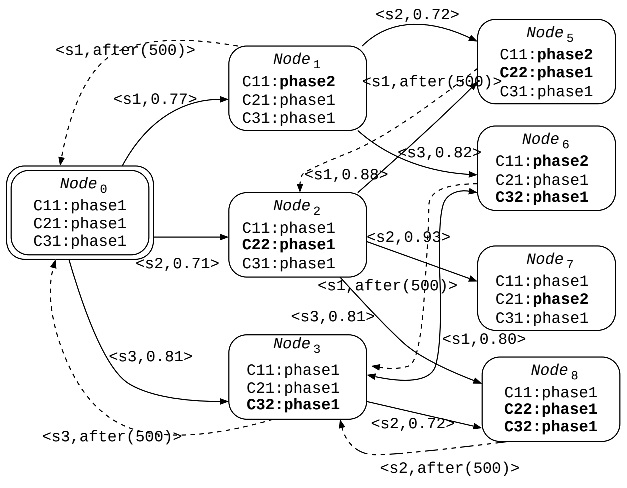

Figure 5: Partial reconfiguration strategy graph

The $Node_1$ is added considering the active provider of service s_1 in $Node_0$ (i.e., c_{11}) changes its *phase* by answering to service requests with a mean service time of $20tu$, instead of $5tu$, i.e. from

$phase_1$ to $phase_2$. 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 of Figure 4 is used to compute the system mean response time of the configuration $Node_1$.

The $Node_2$ is created assuming 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 configurations in the first column of Table 2 with the provider c_{21} in $phase_2$. In the other two configurations, the active provider of s_2 is c_{22} (i.e, the configurations in the second column of 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 consideres 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:

- 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 $phase_1$ to $phase_2$ and the system moves from $Node_0$ to $Node_2$). This is estimated as:

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

  where $rt_{s|c_{kl} \leftarrow phase_{j+1}}$ 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 $phase_j$ to $phase_{j+1}$, 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:

$$conf\_level = \frac{Perf_{improve}}{Perf_{improve} + Perf_{loss}}. \quad (1)$$

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 $conf\_level = \frac{rt_s}{rt_t}$.

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 $phase_{Jkl}$ and in $Node_t$ with its initial $phase_1$. 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 a first attempt to accomplish comprehensively most of such challenges. Also, we have explored how to generate strategies, that can reconfigure this software while performance goals have 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 a case study. 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. In this work, we have not considered network transmission times, however they can be easily incorporated through the UML deployment diagram.

Regarding to the 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 [4], 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 do not deal with the problem of strategy generation. The work of Garlan in [2] also proposes an architecture for performance evaluation but restricted to self-healing systems, besides they do not use of formal methods. Finally, in [9] is proposed an architecture to manage the adaptation for evolvable systems, but this work does not deal with performance evaluation.

7. REFERENCES

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.