Formal Specification and Evaluation of Components Interaction in Software Architectures

Mehran Sharafi [1], Fereidon Shams Aliee [2], Ali Movaghar [3]

[1] Faculty of Engineering, Azad Islamic University of Najafabad, PhD student ,Tehran Science and Research branch, Iran
[2]Faculty of Computer Engineering, Shahid Beheshti University, Tehran, Iran
[3] Computer Engineering Department, Sharif University of Technology, Tehran 11365, Iran
Mehran_sharafi@iaun.ac.ir, f_shams@sbu.ac.ir,Movaghar@sharif.edu

Abstract

Software Architecture refers to a high-level specification of structures and behaviors of software components. This specification can be used for evaluating non-functional attributes, even before software implementation, and therefore it results in early detection and correction of defects. In this way, formal methods are more rigorous and systematic and make it possible to automatically verify different aspects of software architectures. In this paper we introduce a framework to formally specify and evaluation of Software Architectures. The frame work includes an algorithm for transforming Software Architecture described in UML to a powerful and formal model, called Team Automata. The framework also proposes a performance model over obtained formal descriptions. This model is used for specifying, evaluating and enhancing the architecture of a Web-Service software, under flash-crowd condition and the results of analysis and experiments are presented.

Keywords: Software Architecture, Team Automata, Component Interaction, Performance.

1. Introduction

Software Architecture (shortly SA) in early development phases, represents models which contain basic structural components of software and their interactions; on the other hand it contains both static structure and dynamics of system behavior. In spite of very high level of abstraction of architectural models, they comprise important design features which could be used to anticipate functional and non-functional attributes (like performance, security, etc.) of software. In the past several years, many methods for specifying and evaluating SA have been proposed, and their primary goal is to facilitate architectural decisions making; for example to choose a suitable architecture among several architectural alternatives, one that best fit to functional and non-functional requirements of relevant software.[21,20,5,4]

Some of these methods are base on formal models, some are language-based (Architectural Description Languages-ADLs) and the others like UML propose visual descriptions. Each of mentioned methods has their own benefits and drawbacks and making use of them depends on the purpose of architect from describing architecture.

ADLs are suitable for specifying hierarchical structures of components and also interaction between them. Moreover they have tool supports. On the other side, UML diagrams are highly understandable and are widely used by software engineers. UML 2.0 specially has significant extensions for specifying SA. However, the essential drawback of all informal methods versus formals is that their specification power is often not general enough to preserve all the interaction properties which might arise through components composition. Additionally the verification and validation within an informal framework usually supports only a few fixed set of properties. But formally specifying software architectures make a highly formal and general foundation which could be used to verification of different aspects which can be deduced from overall structure and interaction between components. Finite state machines (FSMs), labeled transition system (LTSs) and Petri nets have been widely used in the literature for this purpose [2, 3, 4,5,6,7, 22].

However these models are designed for modeling component interaction and are often unable to describe the interconnection structure of hierarchical component architecture. In this work we introduce a formal framework to specify and evaluate software architectures and try to overcome usual limitation of common formal models. Within the framework we have proposed an algorithm to transform SA behaviors described in UML 2.0 to an automata based model called team automata [8]. In this manner a rigorous foundation for further activities (e.g. verification of non-functional properties) have been developed. Beside the formal descriptions, we proposed a performance model which used to evaluate performance aspects of software architecture. Thus our framework could be used by software architects to choose suitable architecture among many alternatives and/or help them to make changes to an architecture to fit desired performance requirements. This paper organized as follow: After Introduction, in Section 2 a comparison is made between some extended automata-based models, and their abilities and weakness to specify components interaction are described. In this section also Team Automata, as a selected model, will be introduced and some definitions applied in our algorithm explained. In Section 3 we introduce overall framework, transformation algorithm and our performance model in detail. In Section 4, transformation algorithm and performance model will be

applied on two alternative architecture of a web-service software as a case study and the results will be presented. Section 5 refers to conclusions and future works.

2. Using automata-based models to specify SA.

As we mentioned before, automata-based models have been used in the literature to specify dynamics of software architectures. However some of extended automata are more consistent for this issue because they designed for modeling the interaction between loosely coupled components in systems. For example Input/Output Automata (shortly IOA) [9] as a labeled transition system provide an appropriate model for discrete event systems consisting of concurrently operating components with different input, output and internal actions. IOA can be composed to form a higher-level I/O automaton and thus form a hierarchy of components of the system. Interface Automata(IA) [10, 11] are another extended automata model suitable for specifying component-base systems, which also support incremental design. Finally, Team Automata [8] is a complex model designed for modeling both the conceptual and the architectural level of groupware systems.

Common feature of these automata models is that "actions" are classified in 'input', 'output' and 'internal's, such that internal actions can not participate in components interaction. This feature has made them powerful to specify interaction between loosely coupled and cooperating components. It is clear that there are many similarities between application domains of the mentioned models and the literature of Software Architectures. Thus applying these models in SA area must be greatly taken in to consideration by software engineers. In [12] we also made a detailed comparison between these models and described why we have selected Team Automata for our framework.

2-1. Team Automata

Team Automata model was first introduced in [13] by C.A.Ellis. This complex model is primary designed for modeling groupware systems with communicating teams but can be also used for modeling component-based systems [14]. In this section some definitions of TA literature are briefly described. These definitions have been used in the algorithm proposed in this paper. Readers are referred to [8] for more complete and detailed definitions.

Let $\Gamma \subseteq \mathbb{N}$ be a nonempty, possibly infinite, countable set of indices. Assume that Γ is given by $\Gamma = \{i_1, i_2, \ldots\}$, with $i_j < i_k$ if $j < k$. For a collection of sets V_i, with $i \in \Gamma$, we denote by $\prod_{i\in\Gamma} V_i$ the Cartesian product consisting of the elements $(v_{i1}, v_{i2}, \ldots)$ with $v_i \in V_i$ for each $i \in \Gamma$. If $v_i \in V_i$ for each $i \in \Gamma$, then $\prod_{i\in\Gamma} v_i$ denotes the element $(v_{i1}, v_{i2}, \ldots)$ of $\prod_{i\in\Gamma} V_i$. For each $j \in \Gamma$ and $(v_{i1}, v_{i2}, \ldots) \in \prod_{i\in\Gamma} V_i$, we define $\text{proj}_j((v_{i1}, v_{i2}, \ldots)) = v_j$. If $\phi \neq \zeta \subseteq \Gamma$, then $\text{proj}_\zeta((v_{i1}, v_{i2}, \ldots)) = \prod_{j\in\zeta} v_j$.

For the sequel, we let $S = \{C_i \mid i \in \Gamma\}$ with $\Gamma \subseteq \mathbb{N}$ be a fixed nonempty, indexed set of component automata, in which each C_i is specified as $\left(Q_i, \left(\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,\text{int}}\right), \delta^i, I_i\right)$, with $\Sigma_i = \Sigma_{i,inp} \bigcup \Sigma_{i,out} \bigcup \Sigma_{i,\text{int}}$ as set of actions. $\Sigma = \bigcup_{i\in\Gamma} \Sigma_i$ is the set of actions of S and $Q = \prod_{i\in\Gamma} Q_i$ is the state space of S.

Component automata interact by synchronizing on common actions. Not all automata sharing an action have to participate in each synchronization on that action. This leads to the notion of a complete transition space, consisting of all possible combinations of identically labeled transitions.

Definition 1. A transition $(q, a, q') \in Q \times \Sigma \times Q$ is a *synchronization* on a in S if for all $i \in \Gamma$, $(\text{proj}_i(q), a, \text{proj}_i(q')) \in \delta^i$ or $\text{proj}_i(q) = \text{proj}_i(q')$, and there exists $i \in \Gamma$ such that $(\text{proj}_i(q), a, \text{proj}_i(q')) \in \delta^i$.

For $a \in \Sigma$, $\Delta_a(S)$ is the set of all synchronizations on a in S. Finally $\Delta(S) = \bigcup_{a\in\Sigma} \Delta_a(S)$ is the set of all synchronizations of S.

Given a set of component automata, different synchronizations can be chosen for the set of transitions of a composed automaton. Such an automaton has the Cartesian product of the states of the components as its states. To allow hierarchically constructed systems within the setup of team automata, a composed automaton also has internal, input, and output actions. It is assumed that internal actions are not externally observable and thus not available for synchronizations. This is not imposed by a restriction on the synchronizations allowed, but rather by the syntactical requirement that each internal action must belong to a unique component:

S is *composable* if $\Sigma_{i,\text{int}} \cap \bigcup_{j \in \text{T}} \Sigma_j = \phi$ for all $i \in$ I.

Moreover, within a team automaton each internal action can be executed from a global state whenever it can be executed by its component at the current local state. All this is formalized as follows.

Definition 2. Let S be a composable set of component automata. Then a *team automaton* over S is a transition system $\text{T} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{\text{int}}), \delta, I)$, with set of states $Q = \Pi_{i \in \Gamma} Q_i$ and set of initial states $I = \Pi_{i \in \Gamma} I_i$, actions $\Sigma = \bigcup_{i \in \Gamma} \Sigma_i$ specified by $\Sigma_{\text{int}} = \bigcup_{i \in \Gamma} \Sigma_{i,\text{int}}$, $\Sigma_{out} = \bigcup_{i \in \Gamma} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i \in \Gamma} \Sigma_{i,inp}) \setminus \Sigma_{out}$ and transitions $\delta \subseteq Q \times \Sigma \times Q$ such that $\delta \subseteq \Delta(S)$ and moreover $\delta_a = \Delta_a(S)$ for all $a \in \Sigma_{\text{int}}$.

As definition 2 implies, one of the important and useful properties of TA compared with other models is that there is no unique Team automata composed over a set of component automata, but a whole range of Team Automata distinguishable only by their synchronizations can be composed over this set of component automata. This feature enables Team automata to be architecture and synchronization configurable, moreover, makes it possible to define a wide variety of protocols for the interaction between components of a system.

Tow other definitions which effectively used in our algorithm are "subteams" and "communicational actions" that we briefly introduce. Reference,[8] supports detailed definitions.

Definition3. A pair Ci,Cj with i,j$\in \Gamma$, of component automata is *communicating* (in S) if there exist an $a \in (\Sigma_{i,ext} \bigcup \Sigma_{j,ext})$ such that $a \in (\Sigma_{i,inp} \bigcap \Sigma_{j,out}) \bigcup (\Sigma_{j,inp} \bigcap \Sigma_{i,out})$.

Such an a is called a *communicating* action (in S). By Σ_{com} we denote the set of all *communicating actions* (in S).

Definition 4. Let $\text{T} = (\Pi_{i \in \Gamma} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{\text{int}}), \delta, \Pi_{i \in \Gamma} I_i)$ be a team automaton over the composable system S and let $J \subseteq \Gamma$. Then the *subteam* of T determined by J is denoted by $SUB_J(T)$ and is defined as $SUB_J(T) = (\Pi_{j \in J} Q_j, (\Sigma_{j,inp}, \Sigma_{j,out}, \Sigma_{j,\text{int}}), \delta_J, \Pi_{j \in J} I_j)$, where:

$\Sigma_{J,\text{int}} = \bigcup_{j \in J} \Sigma_{j,\text{int}}$,

$\Sigma_{J,out} = \bigcup_{j \in J} \Sigma_{j,out}$,

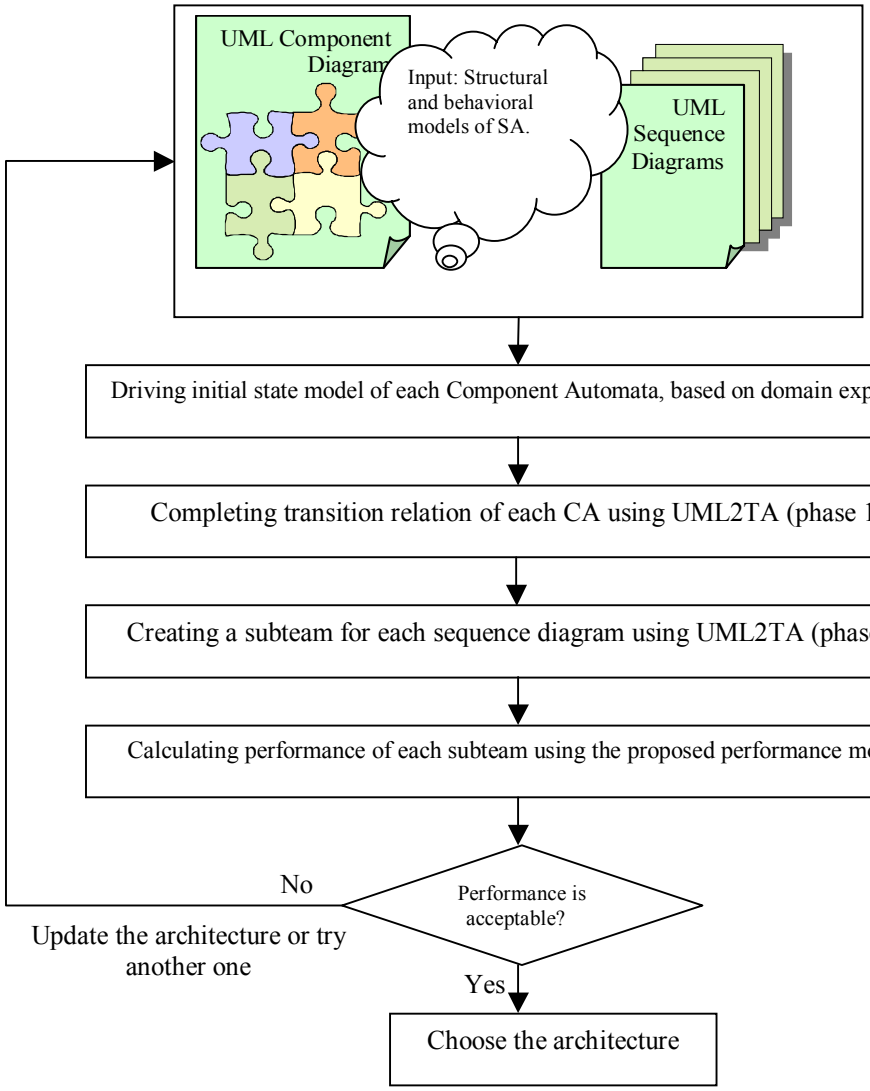
$\Sigma_{J,inp} = (\bigcup_{j \in J} \Sigma_{j,inp}) \setminus \Sigma_{J,out}$ and for all $a \in \Sigma_J = \bigcup_{j \in J} \Sigma_j$,

$(\delta_J)_a = proj_J^{[2]}(\delta_a) \cap \Delta_a(\{C_j | j \in J\})$.

The transition relation of a subteam of T determined by some $J \subseteq \Gamma$, is obtained by restricting the transition relation of T to synchronizations between the components in $\{C_j | j \in J\}$. Hence in each transition of the subteam at least one of the component automata is actively involved. This is formalized by the intersection of $(\delta_J)_a = proj_J^{[2]}(\delta_a)$ with $\Delta_a(\{C_j | j \in J\})$, for each action *a*, as in each transition in this complete transition space at least one component from $\{C_j | j \in J\}$ is active.

3. Proposed Framework.

In this section, we describe an extension made to UML to become consistent and could be used as our input model. Then we introduce an algorithm to transform extended UML models of software architecture to formal descriptions of Team Automata we called this algorithm UML2TA. Finally a performance model is described over TA, to evaluate performance aspects of software architecture. Flowchart of Fig.1. shows overall steps of our framework.



3-1. Input Model

UML diagrams are highly understandable and are widely used by software developers. New version of UML (UML 2.X) have enhanced notations for specifying component-based development and software architectures. [1, 15]

Since our target model-TA, is very formal, direct translation of UML to TA is problematic. Therefore we first provided formal definitions of UML model elements to create a consistent input model. Static structure of software architecture is described with UML 2 Component Diagram, while the interaction between components is described by Sequence Diagrams. Our input model is explained as follow:

A UML Component is defined as uComponent = (Name, PI, RI), where Name is a string, denotes name of the component; PI is a finite set of provide interfaces and RI is a finite set of required interfaces [1] of the component (One can specify a port_id for each interface to identify related port; we ignore definition of ports in our algorithm). Sets, RI and PI are disjoint for a component; however PIs an RIs of two component could have common elements. Each interface has a finite alphabet called *M* which contains names of messages it can exchange with other interfaces. (Messages could be considered as requests (or responds) to a method call through an interface of a component [1]).

A UML Connector is defined as uConnector = (C_1, I, C_2), where: C_1 and C_2 are names of components which are both ends of the connector and *I* is name of the interface involved the connection. If *I* be the name of a required interface (or a provide interface) of both C_1 and C_2 then type of the connector is 'delegate'. However if I be the name of a required interface of C_1 and a provide interface of C_2 then type of the connector is assembly. (We assume that designers or architects follow these constraint at the time of creating UML models).

A UML Component Diagram is defined as uCD= (Components, Connectors), where Components is a finite set of uComponents and Connectors is a finite set of uConnectors.

A UML Sequence Diagram is corresponding to a system scenario and is defined as uSD=(Components,Stimuli), where components is the set of components participating in the scenario and Stimuli is a vector of messages, according to the order of the time in the Sequence Diagram. A message passing is defined as uStimulus=(C,a), where C is a uConnector and *a* is the name of message passed through the interface corresponding to C ($a \in C.I.M$).

3-2. UML2TA - an algorithm for transforming UML description to TA

As mentioned before, SA description contains a UML component diagram (CD) specifying structural feature of SA and a finite set of Sequence Diagram specifying scenarios of system i.e. components interactions. First step in UML2TA is that each component in CD, is considered as a component automata C_i in set $S = \{C_i | i \in I\}$ (Suppose, g:CD.Components→S is a one-to-one correspondence which maps each software component to a component automata C_i). Initially, C_i is manully derived from the informal behavioral description of the component, based on domain expertise; to do this step one can use method introduce in [9]; The goal of this step is to obtain all the possible states in each component automata, not necessary all the transition. Each C_i can be incomplete in trams of transitions. Now we have $C_i = (Q_i, \Sigma_i, \delta_i, I_i)$, where I_i is complete and the rest may not. Names of internal actions of a component (if there are any) must be different from all actions of other component for satisfying 'composability condition' of component automata [8]).

In order to complete the sets of input action ($\Sigma_{i,inp}$) and output actions ($\Sigma_{i,out}$) of each C_i we use CD as follows:

For each Component = (Name, PI, RI), $Component \in CD.Components$, which g(component) = C_i, do as following:

$$\Sigma_{i,inp} = \Sigma_{i,inp} \bigcup Component.PI.M$$

$$\Sigma_{i,out} = \Sigma_{i,out} \bigcup Component.RI.M$$

In order to complete δ_i for each C_i, we find out for each Stimuli of form, ((C_i, I, C_j),a) in each SD, which states of C_i and C_j belongs; then according to precondition and effects of action *a* in both C_i , C_j , if *a* causes a transition from q to q/ in C_i $(q, q' \in Q_i)$, and a transition from s to s/ in C_j $(s, s' \in Q_j)$ then do as following:

$$\delta_i = \delta_i \cup (q, a, q'),\ \delta_j = \delta_j \cup (s, a, s')$$

Now we have set $S = \{C_i | i \in \Gamma\}$, including all the component automata of the system. According to the definition of TA (def. 2), we can compose many team automata over *S* distinguishable by their transition relation, where, each of them is corresponding to a specific components interaction. However, in UML, components interaction is specified by a set of scenario and each scenario could be described by a sequence diagram, therefore we must again use the set, SD to define a specific team automata or a set of subteams, over s. In this way if team automata of the system is defined as $\tau = (\prod_{i \in I} Q_i, (\Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,\text{int}}), \delta, \prod_{i \in I} I_i)$ then we model each sequence diagram SD_k in SD as a subteam of τ ; In other word, each SD_k is modeled as $SUB_{J_k}(\tau)$ and $J_k \subseteq I$; where J_k contains the indices of a subset of component automata participating in the scenario corresponding to SD_k, and:

$$SUB_{J_k}(\tau) = (\prod_{j \in J_k} Q_j, (\Sigma_{J_k,inp}, \Sigma_{J_k,out}, \Sigma_{J_k,\text{int}}), \delta_{J_k}, \prod_{j \in J_k} I_j)$$

Now we have all the above set except for δ_{J_k} . To define δ_{J_k} we do as follow:

Since, for each SD_k.Stimuli=((C_i,I,C_j),a) ,message *a* is sent from C_i to C_j, thus, in our TA model, C_i, executes *a* as an output action and C_j executes *a* as an input action; therefore *a* is a

communicating action [8] and we can model the Stimuli as a synchronization on a. So, to define transition relation of $SUB_{J_k}(\tau)$, we do the following for each Stimuli of form ((C_i,I,C_j),a):

$\delta_{J_k} = \delta_{J_k} \bigcup (q, a, q')$

where $(proj_i(q), a, proj_i(q')) \in \delta_i$ and $(proj_j(q), a, proj_j(q')) \in \delta_j$ such that, $a \in \Sigma_{i,out}$, $a \in \Sigma_{j,inp}$

Since each Stimuli, introduces message passing from a required interface of a component to provide interface of another component, therefore the connection between tow component is an assembly connection [1]. Hence, the message is *consumed* by the destination component and no longer exists for other synchronization outside of the subteam (or team). This explanation is different from definition of team automata in which output actions of all component automata will be output actions of the team over them. So we use *hide* operator [8] to hide those output actions from out side of the subteam. Therefore our model for SD_k is defined by the following formulas:

$$hide_{\Sigma_{J_k,com}}\left(SUB_{J_k}(\tau)\right)$$

Where $\Sigma_{J_k,com}$ is the set of communication action of sub team. This suggestion could also be useful when we want to consider each subteam as a component automata and use it as a building block in an incremental design issue.

Tables 1 and 2, are corresponding to the tow phase algorithm UML2TA for translating UML diagrams to a TA specification.

Table1- Phase 1 of UML2TA to create the set of Component Automata S.

Inputs:

1. A UML Component diagram, D=(Components,Connectores),|CD.Components|=m

2. SD[] is the set of UML sequence diagrams, |SD|=k , SD[i]=(Components, stimui) for each i, $1 \le i \le k$ *. SD[i].Components is the set of all components interacting within SD[i] and SD[i].stimuli is an array of UML stimulus of form ((C_1,I,C_2),a) .*

Outputs:

1.S[] is the set of component automata where S[i](=C_i), is component automaton indexed by i, S[i]=(Q,SIGMA,DELTA,I)

2. g is a one-to-one correspondence between CD.Components and a set of indices [1..m].

Translate(CD,SD):S

i ⟵ *0*

$g \leftarrow \phi$

while CD.Components $\neq \phi$ *do begin*

i ⟵ *i+1*

uC∈ *CD.Components*

S[i] ⟵ *initial state model of uC*

$g \leftarrow g + \{(uC, i)\}$

while uC.PI $\neq \phi$ *do begin*

uPI∈ *uC.PI*

S[i].SIGMA.Inputs ⟵ *uPI.M*

uC.PI ⟵ *uC.PI - {uPI}*

end

while uC.RI $\neq \phi$ *do begin*

uRI∈ *uC.RI*

S[i].SIGMA.Outputs ⟵ *uRI.M*

uC.RI ⟵ *uC.RI-{uRI}*

end

CD.Components ⟵ *CD.Components-{uC}*

```
end
for i:=1 to NUMBER_OF_SDs do begin
        for j:=1 to SD[i].NUMBER_OF_STIMULUS do begin
                if SD[i].stimuli[j].a causes a transition from state q to atate q' in
        component SD[i].Stimuli[j].C.C1 and also it causes a transition from
                state s to s' in component SD[j].stimuli[j].C.C2 such that, g(C1)=1
                g(C2)=j then begin
                        s[i].DELTA ⟵ s[i].DELTA+{(q,a,q')}
                        s[j].DELTA ⟵ s[j]>DELTA+{(s,a,s')}
                end //if
        end
end
End Translate.
```

Table2- Phase 2 of UML2TA to create Subteams.

Input:

1. SD[] is the set of UML sequence diagrams, |SD|=k , SD[i]=(Components, stimui) for each i, $1 \le i \le k$ *. SD[i].Components is the set of all components interacting within SD[i] and SD[i].stimuli is an array of UML stimulus of form* $((C_1,I,C_2),a)$ *.*

2. .S[] is the set of component automata where $S[i](=C_i)$ *, is component automaton indexed by i,*

Output:

1. Team Automata of the system ,T=(Q,SIGMA,DELTA,I).

2. SUB[] is the set of Subteams , such that SUB[i] is correspond to SD[i] and SUD[i]=(Q,SIGMA,DELTA,I), $1 \le i \le k$ *.*

CreateSubTeams(SD,S):SUB

```
For i:=1 to NUM_SD do begin    //Number of sequence diagrams
        J ⟵ indices of all components in SD[i].Components
        SUB[i].Q ⟵ Cartesian product of s[j] for all j∈ J
        SUB[i].SIGMA ⟵ Union of actions of component automata S[j], for all j∈ J
//composability should be satisfied
SUB[i].I⟵ Cartesian product of S[i].I for all j∈ J
End
T ⟵ CREATE_TEAM(SUB); // includes all the Subteams that would be completed as follow
for i:=1 to NUM_SD do begin
        temp1 ⟵ SUB[i].I.CurrentState
        for j:=1 to SD[i].NUM_STIMULS do begin
                If j<>1 then begin
                        r ⟵ g(SD[i].Stimuli[j].C.C1)
                        if find a transition SGM in S[r] such that:
                                (s[r].SGM.q1=proj(r,temp1)
                                and S[r].SGM.a=SD[i].Stimuli[j].a   //q1 is a_enabled
                                and s[r].SGM.a in S[r].SIGMA.Outputs)
                        then    Proj(r,temp2) ⟵ s[r].SGM.q2
                        else Error
                end
                r ⟵ g(SD[i].Stimuli[j].C.C2)
                if find a transition SGM in S[r] such that:
                                (s[r].SGM.q1=proj(r,temp1)
                                and S[r].SGM.a=SD[i].Stimuli[j].a   //q1 is a_enabled
                                and s[r].SGM.a in S[r].SIGMA.Inputs)
                        then    Proj(r,temp2) ⟵ s[r].SGM.q2
                        else Error
                NEWSGM ⟵ MakeString(temp1,SD[i].stimuli[j].a,temp2)
                SUB[i].DELTA ⟵ SUB[i].DELTA+{NEWSGM}
                Temp1 ⟵ temp2
        end
end
End_ CreateSubTeams
```

3-3. A performance model over TA specification.

Until now we develop a formal foundation for software architecture which can be used for evaluating several attributes (For example in [23], [24] TA have been used to security analysis of groupware systems). In this section we introduce a model to evaluate performance of software architecture described by team automata. In this way two features have been considered for evaluating performance:

a) Performance specifications of components communication. This feature depends on a variety of factors e.g. components deployment, centralization/distribution, network quality and so on. This information are not explicitly given in the architecture descriptions; to obtain such information, one can use data collected from similar existing systems. Newer versions of UML facilitates specifying performance data within architectural diagrams [15] our UML definitions of Connectors (Section 3) also allow to involve communication delay of each connection (as an extension suppose, each connector could be considered as a quadruple (C_1, I,C_2,d) where d is the corresponding delay). In our performance model, we consider a delay for each synchronization within a subteam.

b) The granularity of the performance analysis. Performance can be analyzed either behavior-dependent or behavior-independent. For example, performance can be defined by processing time of the entire component or processing time of each service invocation in the component. In our model performance is considered at the service level. Since in our model, service requests to a software component assume to be input actions to corresponding component automata, we assign a processing time to each input action. (These data are again obtained from existing similar systems). According to suggestions a and b, we can extend team automata models to include performance information as follows:

For each Component Automata a processing-time function $P:\sum_{i,inp}\longrightarrow R^{+}$ and a delay function $P':\delta_i\longrightarrow R^{+}$ is defined as follow:

P= $\{(a,r)\,|\,a\in\sum_{i,inp}$, r is the processing time corresponding to action $a\}$

P'= $\{(\theta,d)\,|\,\theta\in\delta_i$, d is the delay corresponding to transition $\theta\}$

We now model each Component Automata in the architecture with the extension of performance model as follow:

$$CP_i=((Q_i,(\sum_{i,inp},\sum_{i,out},\sum_{i,\mathrm{int}}),\delta_i,I_i),P_i,P_i')$$

Delays of transition within a component could be ignored (comparing with communication delay between components, especially for distributed components). If we assume components interactions synchrony and sequential then we can consider a whole subteam as a complex server [25] whose mean service time is equal to summation of service time of input actions (those which are synchronized) plus all synchronization delay in the sub team. Thus if $\theta_i\in\delta_{J_k}$ be the ith synchronization in $SUB_{J_k}(\tau)$ and $\Sigma_{J_k,com}$ be the set of all communicating action in $SUB_{J_k}(\tau)$ and $A\subseteq\Sigma_{J_k,com}$ (, $A=\{a_1,a_2,...,a_m\}$, $m=|\delta_{J_k}|$) be the set of communication actions which are synchronized within $SUB_{J_k}(\tau)$ then we have:

$$\frac{1}{\mu_k}=\sum_{i=1}^{m}(P'(\theta_i)+P(a_i))$$

, Where $\frac{1}{\mu_k}$ is mean service time of scenario k (correspond to SD_k) which has been modeled by subteam, $SUB_{J_k}(\tau)$.

Now suppose that software has k independent scenario whose probability of request by users is f_k and λ be total input rate in of requests to the system. (When a request arrives while a previous request of the scenario has not been answered, the new request will be queued). The system response

time corresponding to architecture under evaluation is equal to $R = \frac{1}{\lambda - \mu}$; where μ is total service rate and is calculated by the following formulas:

$$\frac{1}{\mu} = \sum_{i=1}^{k} \frac{f_i}{\mu_i}$$

4. An application system example

We evaluated UML2TA method on a part of a web-service software architecture. In this example we have a component diagram describing major components and connectors (Fig 2), and a sequence diagram (Fig 3) describing components interaction corresponding to a scenario where some end user requests the web content available from /ping URL (This system have been used as a case-study in [17] in a different scope). We use extension defined in [18] for sequence diagrams.

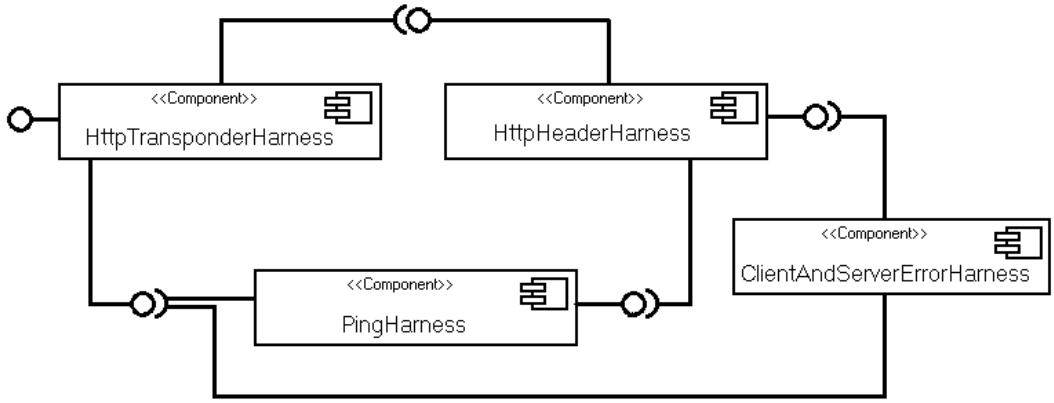


Fig2 .Component Diagram of a part of Web-Service Software.

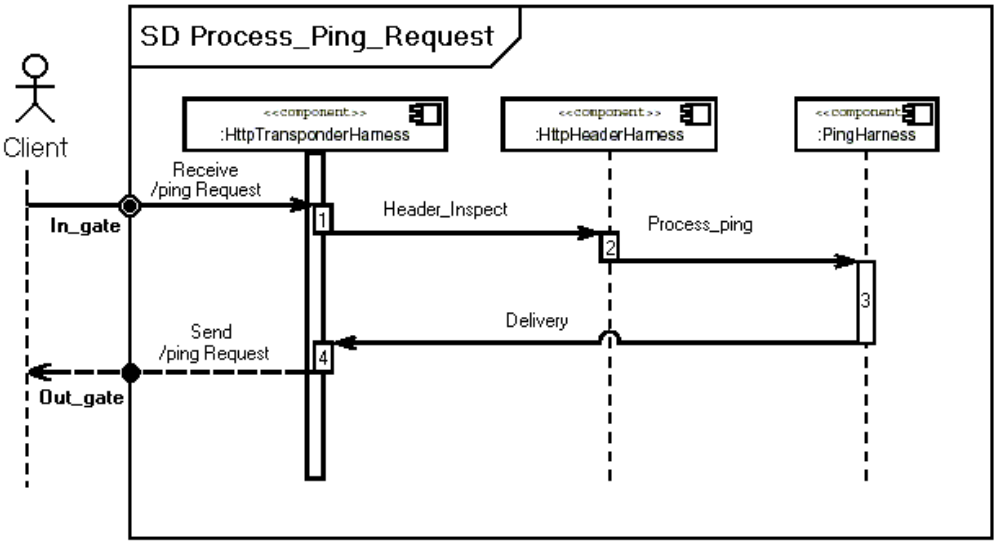


Fig.3. Sequence Diagram specifying components interaction for '/ping' Scenario.

The behavior of components is briefly discussed as follow. The HttpTransponderHarness component is responsible for all client communication (both request receipt and response generation). The component HttpHeaderHarness interprets header of the message and identify type of request. Finally PingHarness generates response HTML text that will eventually echo the request text back to the client in their browser. In the region 1 of Figure 1, a network client sending a HTTP request for a '/ping' requests to server socket that the HTTPTransponderHarness is listening with. The component allocates a new thread from its thread pool that collects the request data from the client socket once data is received it sends a message to HttpHeaderHarness to parse the header of the message. In the region 2, if there is a problem with the HTTP header, HttpHeaderHarness, delivers the request back to the space, marked up to trigger an error processing (not shown in diagram) to take the request, and

generate an appropriate error response for delivery back to the requesting client. The request in this scenario, however, has a valid HTTP header and indicates that the intended resource desired from the HTTP request is identified by the '/ping' URL fragment. In the region 3, PingHarness component receives a message identifying a /ping request, then it generates response HTML text that will eventually echo the request text back to the client in their web browser. Region 4 is where the HttpTransponderHarness receives a message from PingHarness, and transmit the response content of the request back along original client socket for this request in a newly allocated thread pool.

4-1. TA models of the Web-Service Software Architecture.

According to UML2TA, First, we manually model each software component with a Component Automata from informal behavioral descriptions which briefly mentioned.

Table 3 describes Component Automata of each software component.

Table3. CA models of Web-service Software components .

Component Automata model of HttpTransponderHarness:	Component Automata model of HttpHeaderHarness	Component Automata model of PingHarness
Actions: Input actions : /ping_req , delivery. Output actions: /ping_resp , header_inspect. Internal action: new_thread_allocation. State Variables: Process_Input :{0,1} Prepare_resp: {0,1} Transitions(per actions): /ping_request: Effect: process_inp := 1; delivery: Effect: prepare_resp := 1; /ping_resp: Preconditions: prepare_resp:=1; Effects: prepare_resp:=0; /header_inspect: Preconditions: process_inp:=1; Effects: process_inp := 1;	Actions: Input actions: header_inspect; Output actions: proc_ping; Internal action: none; State Variables: Identify_request_type : {0,1}; Transitions: (per actions) header_inspect: Effects: Identify_request_type := 1; proc_ping: Preconditions: Identify_request_type := 1; Effects: Identify_request_type := 0;	Actions: Input action: proc_ping; Output action: delivery; Internal action: None; State Variables: Generate_response:{0,1}; Transitions (per actions): Proc_ping: Effects: generate_response := 1; delivery: Preconditions: generate_response := 1; Effects: generate_response := 0;

If we have all scenarios of the system then we can model TA of overall system; However according to algorithm UML2TA, for each scenario we can create a subteam; Therefore if components HTTPTransponderHarness, HttpHeaderHarness and PingHarness be corresponding to component automata C_1, C_2 and C_3 respectively, then we have:

$$SUB_J(\tau) = \left(Q_J, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{\text{int}}), \delta_J, \prod_{j \in J} I_j \right) \text{where } J = \{1,2,3\},\ Q_J = \prod_{j \in J} Q_j ,$$

$$\Sigma_{inp} = \{/\ ping\_req\} ,$$

$$\Sigma_{out} = \{/\ ping\_resp, Header\_insp, proc\_ping, delivery\},$$

$$\Sigma_{\text{int}} = \{new\_thread\},$$

$$Q_J = \{(w, w', w''), (w, w', gr), (w, I, w''), (w, I, w), (pi, w', w''), (pi, w', gr),$$
$$(pi, I, w''), (pi, I, gr), (po, w', w''), (po, w', gr), (po, I, w''), (po, I, w'')\}$$

and briefly we have:

$$\delta_J = \{((w, w', w''), / ping\_req, (pi, w', w'')), ((pi, w', w''), / Header\_insp, (w, I, w'')),$$
$$..., ((po, w', w''), / ping\_resp, (w, w', w''))\}$$

4-2. Performance evaluation and architectural changes

In Section 4-1 UML2TA was applied on Web-Service Software Architecture and relevant component automata and subteam was generated. In this section we represent results of applying UML2TA on a different version of previous architecture, and show how an architect can choose more suitable architecture regarding over load condition using our framework. Before that, we briefly explain overload and flash crowd conditions in systems especially in web.

In web service prevision it is possible for the unexpected arrival of massive number of service requests in a short time periods, this situation referred to as a flash crowd. This is often beyond the control of the service provider and have the potential to severely degrade service quality and, in the worst case, to deny service to all clients completely. It is not reasonable to increase the system resources for short-time flash crowd events. Therefore if Web-Service Software could detect flash crowds at runtime and changes its own behavior proportional to situation occurred, then it can resolve this bottle neck. In the new architecture, a component has been added to previous one, i.e. PingFactoryHarness; it controls response time of each request, detects the flash crowd situation and directs PingHarness to change its behavior proportional to condition occurred. At the end of this section, results of analysis of both architectures are presented and it is shown that how the new architecture is more effective than old one, to face with flash crowds. Thanks to Lindsey Bradford for giving us the initial performance data of the system.

Fig.4. shows component diagram along with performance data and the new component PingFactoryHarness. We used notations defined in [15] by OMG Group.

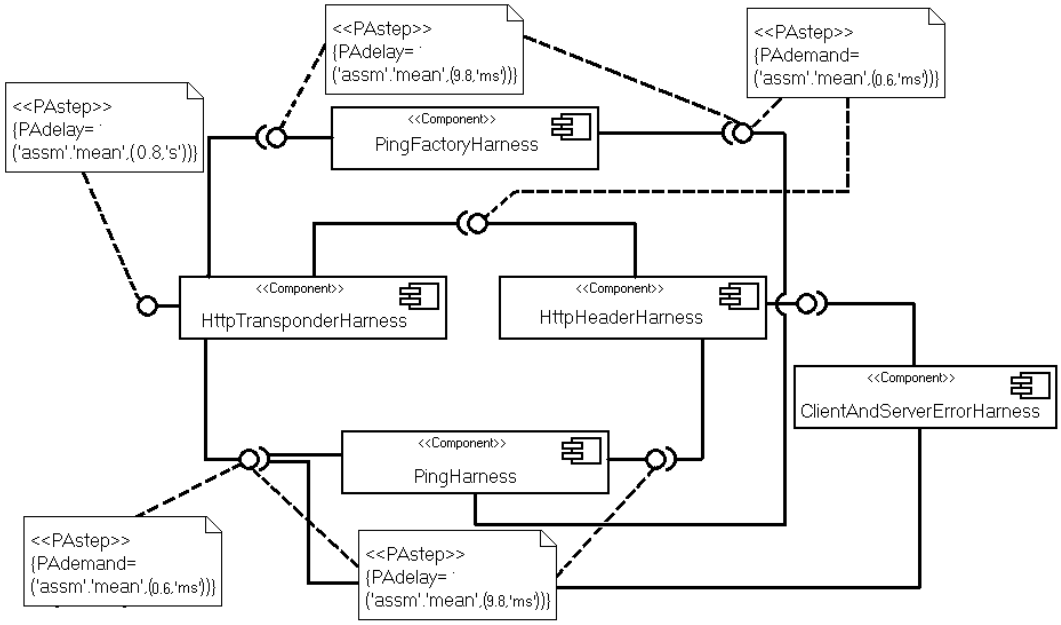


Fig. 4. Extended Component Diagram of new Web-Service Software architecture.

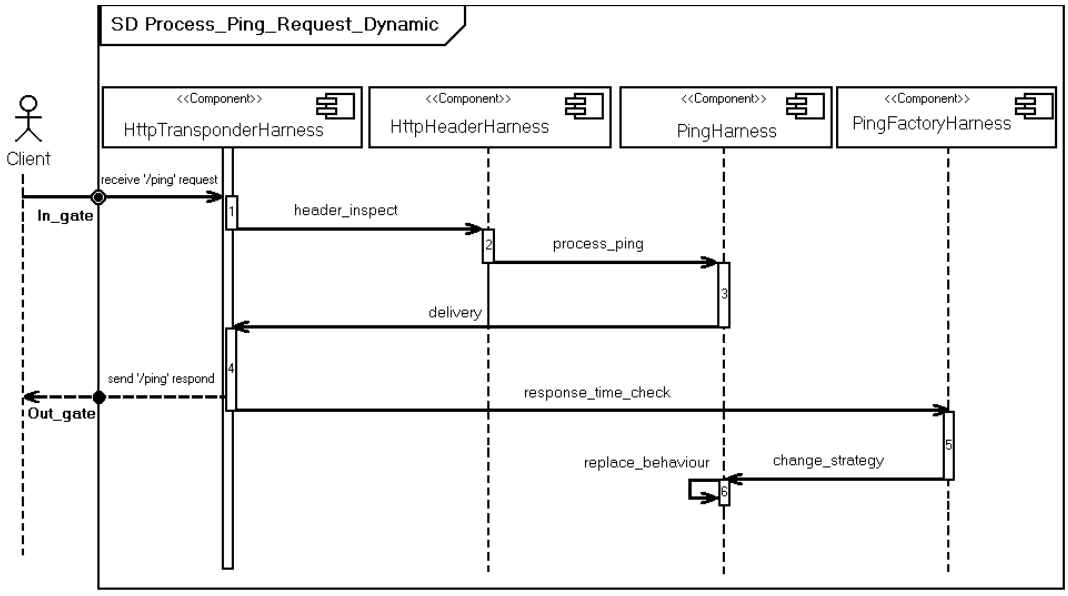


Fig. 5.Sequence Diagram of '/ping' scenario in the new architecture.

Fig 5 shows sequence diagram of '/ping' request in new architecture. Region 1 and 4 execute the same behavior as previous architecture, with a small difference: HttpTransponderHarness takes a snapshot of the system time just after the request text has been received and just before that text is sent to client. This snapshot data used to calculate an elapsed time for responding to the request later in sequence and finally to detect abnormal conditions (e.g. flash crowd). Region 2 is the same as before. The component PingHarness (region 3) is an updated component; it uses a different mechanism to generate response HTML text, and has the ability of changing its behavior when receive relevant message from PingFactoryHarness (We ignore details description due to space limitation).

Region 5 represents behavior of new component: PingFactoryHarness receives the elapse time from HttpTransponderHarness and decides if change is need to the behavior of PingHarness. In the region 6 PingHarness receives the direction of changing behavior.

For the sake of performance evaluation, in experiments performed on both architecture models, in an overload condition, we observe that service times is not stable. It is because of sudden increase of requests for the system resources. This situation dose not follow the flow balancing condition in usual queuing models [16], thus formulating an analytic approach covering the situation is problematic. Hence we use simulation for this part of work and the results of the simulation were used to calibrate analytic model introduce in section 3-4. We summarized the results of our hybrid method to Tables 4 and 5 for the original and updated architecture respectively.

Table 4. Performance data of the old architecture.

Request per Sec.	Response time(ms)			Average number of response per Sec.
	Avg.	Min.	Max.	
2	285.9	284.8	373.9	2
3	1906.3	305.5	7843.5	0.5
5	2877.8	428.8	7744.6	0.2
10	1180.2	1011.2	1397.5	0.0

Table5. Performance data of updated architecture.

Request per Sec.	Response time(ms)			Average number of response per Sec.
	Avg.	Min.	Max.	
2	223.2	222.2	270.8	2
3	229.9	222.3	241.2	3.1
5	7478.1	239.1	10673	3
10	8683.4	255.7	10706	3.4

The difference between the tow architectures at the request rate of 10 per second is interesting. At first glance, it seems that the first architectures response times are much better than the second, However, Comparing throughput between both architectures indicates that first architecture delivered almost no responses at request rate higher than 5; in contrast the second architecture continued to deliver responses, despite the worse response time.

5. Conclusion and Future Works

In this paper, a framework was introduced to formally specify and evaluate Software Architectures. SA specification is initially described in UML2.0 that is the input model for a transformation algorithm called UML2TA introduced within our framework. UML2TA transforms SA descriptions in UML2.0 to a formal model called Team Automata (TA). TA is inspired by Input/Output Automata and has been used in the literature for modeling components interaction in groupware systems. It has also a great generality and flexibility to specify different aspects of components interaction, so it could be best fit to model dynamics of SA. By modeling SA with a powerful model such as TA, a rigorous basis emerged to evaluate (and also verify) functional and non-functional attributes of SA. So we extended usual TA model to include performance aspects which could be involved in UML2.0 diagrams. We also proposed a hybrid performance evaluation model over TA specifications. Finally we applied our framework to the architecture of a web-service software and showed how the framework could be used practically to anticipate performance aspects of an architecture.

In the future works, we decide to firstly, promote our performance model to support wide variety of interactions such as asynchronous, anonymous in distributed environments. Secondly, we are going to enhance our framework to include another non-functional attributes e.g. security; this issue will facilitate simultaneous evaluation of several attributes regarding their conflicting natures.

Refrences

[1] Ivers, P. Clements, D. Garlan, R Nord, B. Schmerl, J. R. Oviedo Silva. *Documenting Component and Connector Views with UML2.0*. Technical report, CMU/SEI, TR-008 ESC-TR-2004-008, 2004.

[2] R. Allen, D. Garlan, A formal basis for architectural connection, ACMTrans. Softw. Eng. Methodol. 6 (3) (1997) 213–249.

[3] D. Giannakopoulou, J. Kramer, S.C. Cheung, Analysing the behaviour of distributed systems using tracta, Automated Software Engineering (special issue), Automated Anal. Softw. 6 (1) (1999) 7–35.

[4] J.J.Li , J.R. Horgan , *Applying formal description techniques to software architectural design,* The journal of Computer Communications, 23,1169-1178, 2000.

[5] M. Shaw, D. Garlan, Software Architecture—Perspectives on an Emerging Discipline, Prentice Hall, Englewood cliffs, NJ, 1996.

[6] R. Allen, R. Douence, D. Garlan, Specifying and analyzing dynamic software architectures, in: Proceedings of FASE , 1998, pp. 21–37.

[7] P.Inverardi and L.Mostarda, *A Distributed Approach for Secure Software Architecture,* R. Morrison and F. Oquendo (Eds.): EWSA 2005, LNCS 3527, pp. 168–184, 2005.Springer-Verlag Berlin Heidelberg 2005.

[8] M. Beek, C. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team Automata for Groupware Systems. Computer Supported Cooperative Work—The Journal of Collaborative Computing, 12(1):21–69, 2003.

[9] N. A. Lynch and M. R. Tuttle. *An introduction to input/output automata*. CWI Quarterly, 2(3):219–246, September 1989.

[10] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. In Volker Gruhn, editor, Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01), volume 26, 5 of Software Engineering Notes, pages 109–120. ACM Press, September 10–14 2001.

[11] Luca de Alfaro and Thomas A. Henzinger. Interface-Based Design. In Proceedings of the Marktoberdorf Summer School, Kluwer, Engineering Theories of Software Intensive Systems, 2004.

[12] M. Sharafi, F Shams Aliee, A. Movaghar. A Review on Specifying Software Architectures Using Extended Automata-Based Models, in proceeding of IPM International Symposium on Fundamentals of Software Engineering (FSEN07), 2007.

[13] C. Ellis. *Team Automata for Groupware Systems*. In Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97), pages 415–424. ACM Press, New York, 1997.

[14] Lubo¡s Brim, Ivana Cern , Pavl´ýna Va¡rekov, Barbora Zimmerova , *ComponentInteraction Automata as a Verification Oriented Component-Based System Specification*, 2005.

[15] Object Management Group. *UML Profile, for Schedulability, Performance, and Time*. OMG document ptc/2002-03-02, http://www.omg.org/cgi-bin/doc?ptc/2002-03-02.

[16] K. Kant and M.M. Sirinivasan. Introduction to Computer Performance Evaluation, McGrawhill Inc. 1992

[17] Lindsay William Bradford. *Unanticipated Evolution of Web Service Provision Software using Generative Object Communication.* Final report of PhD thesis , Faculty of Information Technology Queensland University of Technology, GPO Box 2434, Brisbane Old 4001, Australia, 10 May, 2006.

[18] A. Di Marco, P. Inverardi. Compositional Generation of Software Architecture Performance QN Models Dipartimento di Informatica University of L'Aquila Via Vetoio 1, 67010 Coppito, L'Aquila, Italy, 2004.

[19] N. Medvidovic, R. Taylor, Sclassification and comparison framework for software architecture description languages, IEEE Transaction on Software Engineering 26(1)(2000) 70-93.

[20] L. Bass, P. Clements, R. Kazman, Analyzing development qualities at the architectural level, in: Software Architectures in Practice, SEI Series in Software Engineering, Addison-Wesley, Reading, MA, 1998.

[21] K. Cooper, L. Dai, Y. Deng, Performance modeling and analysis of software architectures: An aspect-oriented UML based approach. Science of Computer Programming, Elsevier ,2005.

[22] X. He, Y. Deng, *A Framework for Developing and Analyzing Software Architecture Specification in SAM*, The Computer Journal vol. 45, No. 1, 2002.

[23] Maurice H. ter Beek, Gabriele Lenzini, Marinella Petrocchi, *Team Automata for Security– A Survey* –,Electronic Notes in Theoretical Computer Science, 128 (2005) 105–119.

[24] L. Egidi, M. Petrocchi, *Modelling a Secure Agent with Team Automata*, The Journal of Electronic Notes in Theoretical Computer Science 142 (2006) 111–127.

[25] Federica Aquilani, Simonetta Balsamo , Paola Inverardi, *Performance analysis at the software architectural design level,* Performance Evaluation 45, Elsevier, (2001) 147–178.