Using Team Automata to Specify Software Architectures

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

[1] Faculty of Engineering, Azad Islamic University of Najafabad
[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. Using formal methods to specify software architectures make it possible to form a rigorous foundation to automatically verify different aspects of software architectures. In this paper we introduce a framework to formally specify and evaluate software architectures. The framework includes an algorithm to transform software architecture described in UML to a powerful and formal model, called Team Automata. The framework also proposes a performance model over the obtained formal descriptions. This model is used to specify, evaluate and enhance the architecture of a Web-Service software under flash-crowd condition and the results of analyses and experiments are presented.

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

1 Introduction

Software Architecture (SA in short) 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 the system behavior. Despite 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 have been proposed to specify and evaluate SA and their primary goal is to facilitate architectural decision-makings; for example, in order to choose a suitable architecture among several architectural alternatives, one that best fits functional and non-functional requirements of relevant software [2, 3, 4, 5].

In this work, we introduce a formal framework to specify and evaluate software architectures and try to overcome the usual limitations 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]. Along by

A. Gelbukh, S. Suárez, H. Calvo (Eds.)
Advances in Computer Science and Engineering
Research in Computing Science 29, 2007, pp. 149-160

Received 30/08/07
Accepted 19/10/07
Final version 24/10/07

the formal descriptions, we proposed a performance model which is used to evaluate performance aspects of software architecture. Thus, our framework could be used by software architects to choose suitable architecture from among many alternatives and/or help them to make changes to architecture to fit desired performance requirements. This paper organized as follows: After Introduction, in Section 2 a comparison is made between some extended automata-based models, and their capabilities and weaknesses to specify components interaction. In this section also Team Automata, as a selected model, has been introduced and some definitions applied in our algorithm have been explained. Section 3 introduces overall framework. In Section 4,the proposed framework, have been applied on two alternative architectures of a web-service software as a case study, and the results have been presented. Section 5 refers to conclusions and future work.

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 have been designed for modeling the interaction among loosely coupled components in systems. For example, Input/Output Automata (IOA in short) [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 thus forming a hierarchy of components of the system. Interface Automata (IA) [10, 11] are another extended automata model suitable for specifying component-based systems, which also support incremental design. Finally, Team Automata [8] is a complex model designed for modeling both the conceptual and architectural level of groupware systems.

The common feature of these automata models is that "actions" are classified in 'inputs', 'outputs' and 'internals', so that internal actions cannot participate in components interaction. This feature has made them powerful to specify interaction among 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 into consideration by software engineers. In [12] we also made a detailed comparison among 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 primarily designed for modeling groupware systems with communicating teams but can also be 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 $\mathrm{proj}_j((v_{i1}, v_{i2}, \ldots)) = v_j$. If $\phi \neq \zeta \subseteq \Gamma$, then $\mathrm{proj}_\zeta((v_{i1}, v_{i2}, \ldots)) = \prod_{j\in\zeta} v_j$.

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 $(Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,\mathrm{int}}), \delta^i, I_i)$, with $\Sigma_i = \Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,\mathrm{int}}$ as set of actions and $\Sigma_{i,ext} = \Sigma_{i,inp} \cup \Sigma_{i,out}$ is the set of external actions of C_i. $\Sigma = \bigcup_{i\in\Gamma} \Sigma_i$ is the set of actions of S; also we have $Q = \prod_{i\in\Gamma} Q_i$ as 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 $i \in \Gamma$, $(\mathrm{proj}_i(q), a, \mathrm{proj}_i(q')) \in \delta^i$ or $\mathrm{proj}_i(q) = \mathrm{proj}_i(q')$, and there exists $i \in \Gamma$ such that $(\mathrm{proj}_i(q), a, \mathrm{proj}_i(q')) \in \delta^i$.

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 set of 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 $\Sigma_{i,\mathrm{int}} \cap \bigcup_{j\in\Gamma\setminus\{i\}} \Sigma_j = \phi$ for all $i \in \Gamma$.

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 $T = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{\mathrm{int}}), \delta, I)$, with set of states $Q = \prod_{i\in\Gamma} Q_i$ and set of initial states $I = \prod_{i\in\Gamma} I_i$, actions $\Sigma = \bigcup_{i\in\Gamma} \Sigma_i$ specified by $\Sigma_{\mathrm{int}} = \bigcup_{i\in\Gamma} \Sigma_{i,\mathrm{int}}$, $\Sigma_{out} = \bigcup_{i\in\Gamma} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i\in\Gamma} \Sigma_{i,inp}) \setminus \Sigma_{out}$ and transitions $\delta \subseteq Q \times \Sigma \times Q$ such that $\delta \subseteq \Delta(S)$ and moreover $\delta_a = \Delta_a(S)$ for all $a \in \Sigma_{\mathrm{int}}$.

As definition 2 implies, one of the important and useful properties of TA compared to 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, it makes possible to define a wide variety of protocols for the interaction among components of a system.

Two other definitions 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 exists an $a \in (\Sigma_{i,ext} \Upsilon \Sigma_{j,ext})$ such that $a \in (\Sigma_{i,inp} \cap \Sigma_{j,out}) \Upsilon (\Sigma_{j,inp} \cap \Sigma_{i,out})$.

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 $T = (\Pi_{i\in\Gamma} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{\text{int}}), \delta, \Pi_{i\in\Gamma} I_i)$ 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:

$\Sigma_{J,\text{int}} = \Upsilon_{j\in J} \Sigma_{j,\text{int}}, \Sigma_{J,out} = \Upsilon_{j\in J} \Sigma_{j,out}, \Sigma_{J,inp} = (\Upsilon_{j\in J} \Sigma_{j,inp}) \setminus \Sigma_{J,out}$

and for all $a \in \Sigma_J = \Upsilon_{j\in J} \Sigma_j$, $(\delta_J)_a = proj_J^{[2]}(\delta_a) \cap \Delta_a(\{C_j | j \in J\})$.

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 among 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. Fig.1. shows the input models and the overall steps of our framework.

3-1. UML2TA: An algorithm for transforming software architecture to Team Automata.

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

Since our target model-TA, is highly 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 among components is described by Sequence Diagrams. Because of space limitation, we ignore describing details of the algorithm (UML2TA) and formal descriptions which we added to initial UML models. Readers are referred to [20] for a complete explanation of our framework. However, in this paper a comprehensive example of applying our framework on a casestudy will be described.

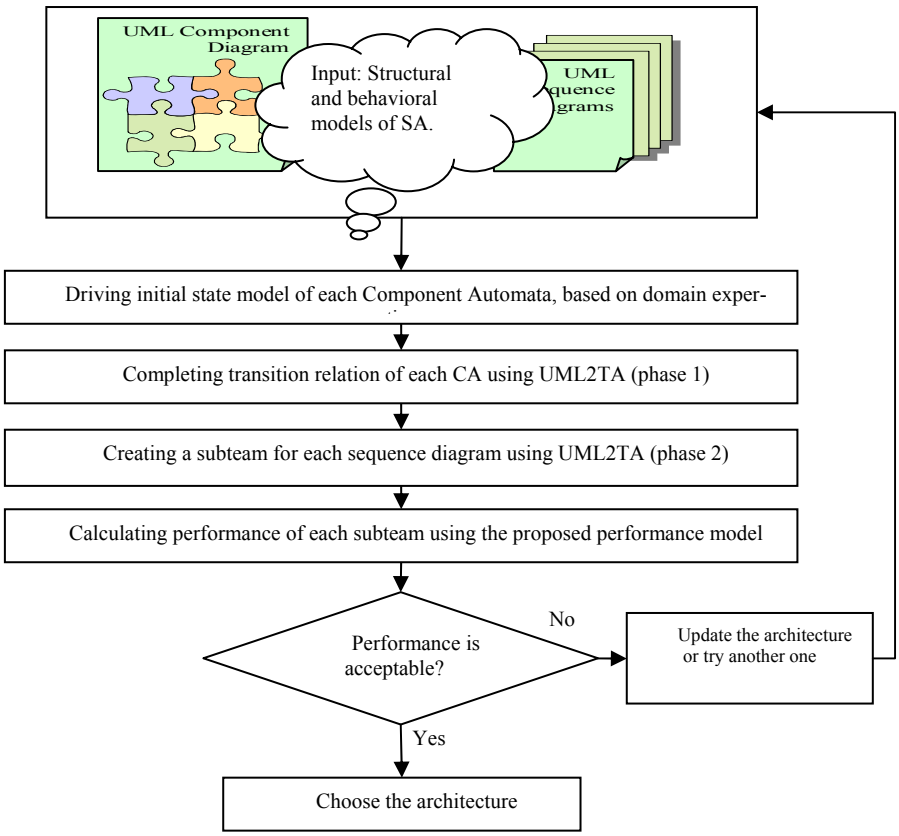

Fig. 1. Overall steps in the framework to formally specify and evaluate software architecture.

3-2. A Performance Model over TA Specifications

TA model achieved by UML2TA, is a formal foundation for software architecture which can be used for evaluating several attributes (For example in [6], [7] TA has been used for 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. In our performance model, we have considered a delay for each synchronization within a subteam.

b) The granularity of the performance analysis. Performance can be analyzed as 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 service requests to a software component are assumed 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 and a delay function P' is defined as follows:

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

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

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

$$CP_i = ((Q_i, (\sum_{i,inp}, \sum_{i,out}, \sum_{i,\text{int}}), \delta_i, I_i), P_i, P_i') \quad \textbf{(1)}$$

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 [19] whose mean service time is equal to summation of service time of input actions (those which are synchronized) plus all synchronization delay in the subteam. 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 actions 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:

$$\frac{1}{\mu_k} = \sum_{i=1}^{m} (P'(\theta_i) + P(a_i)) \quad \textbf{(2)}$$

, Where $\frac{1}{\mu_k}$ is mean service time of scenario k (corresponding 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 suppose, λ is the total input rate of requests to the system (When a request for a scenario arrives while a previous one has not been answered, the new request will be queued). The system response time corresponding to architecture under evaluation is equal to $R=1/(\lambda-\mu)$; where μ is total service rate and is calculated by the following formulas:

$$\frac{1}{\mu} = \sum_{i=1}^{k} \frac{f_i}{\mu_i} \qquad (3)$$

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 has been used as a case-study in [17] in a different scope). We use extension defined in [18] for sequence diagrams.

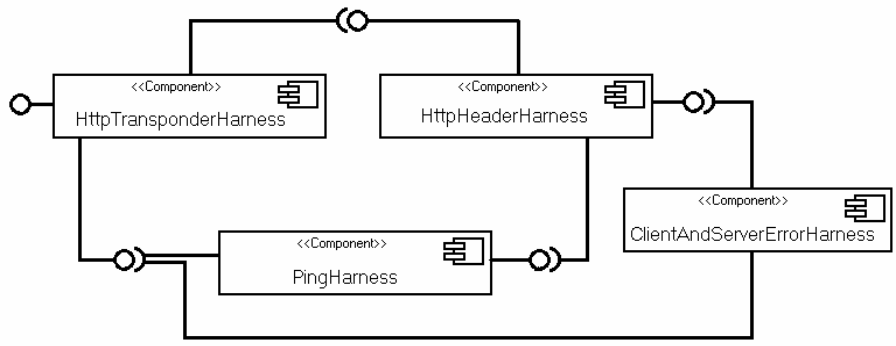

Fig. 2. Component Diagram of a part of Web-Service Software

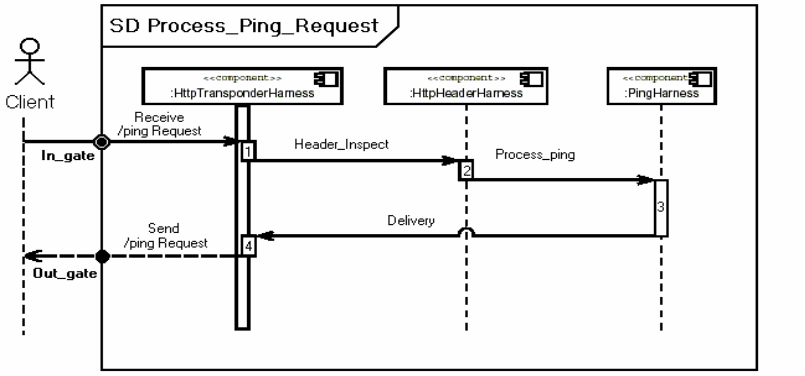

Fig. 3. Sequence Diagram specifying components interaction for '/ping' Scenario.

According to UML2TA, first, we manually model each software component with a Component Automata from informal behavioral descriptions which has been briefly mentioned in Table 1.

Table 1. 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 the overall system; However according to algorithm UML2TA, for each scenario we can create a sub-team; therefore if components HTTPTransponderHarness, HttpHeaderHarness and PingHarness correspond to component automata C_1, C_2 and C_3, respectively, then we have:

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

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

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

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

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

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 were 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 overload condition using our framework. Before that, we briefly explain overload and flash crowd conditions in systems especially in web.

In web service provision, it is possible for the unexpected arrival of massive number of service requests in a short period; this situation is referred to as a flash crowd. This is often beyond the control of the service provider and has 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 change its own behavior proportional to occurred situation, then it can resolve this bottleneck. In the new architecture, a component has been added to the 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 occurred condition. At the end of this section, results of analysis of both architectures are presented and it is shown how the new architecture is more effective than the old one facing 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 have used notations defined in [15] by OMG Group.

In new architecture (sequence diagram of the new scenario has been ignored) HttpTransponderHarness takes a snapshot of the system time just after the request text has been received and just before that text is sent to the client. This snapshot data is used to calculate an elapsed time for responding to the request later in sequence and finally to detect abnormal conditions (e.g. flash crowd. The component PingHarness is an updated component; it has the ability to change its behavior when it receives relevant message from PingFactoryHarness. PingFactoryHarness receives the elapse time from HttpTransponderHarness and decides if change is needed in the behavior of PingHarness. PingHarness then receives the direction to change behavior.

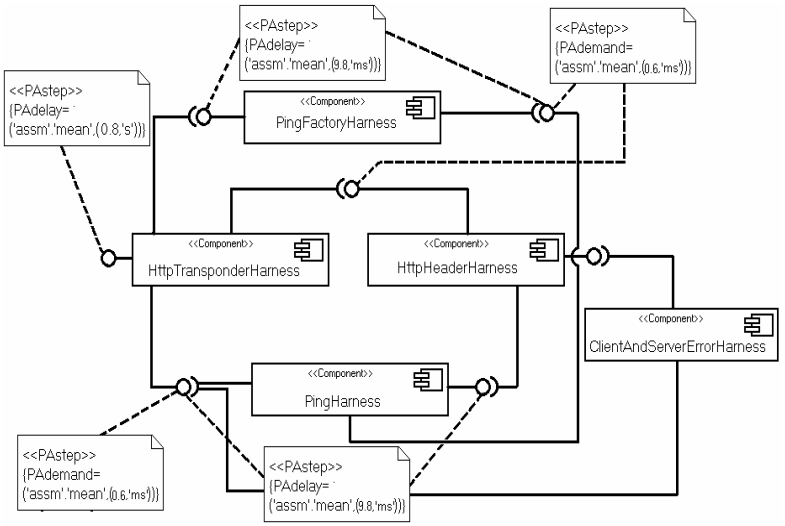

Fig. 4. Extended Component Diagram of new Web-Service Software architecture.

In experiments performed on both architecture models, in an overload condition, we observed that service times are not stable. It is because of sudden increase in requests for the system resources. This situation does 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 introduced in Section 3-2. We summarized the results of our hybrid method to Tables 2 and 3 for the original and updated architecture, respectively.

Table 2. Performance data of the first architecture.

Request per Sec.	Response time(ms)			*Average number of responses 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*

Table 3. Performance data of updated architecture.

Request per Sec.	Response time(ms)			*Average number of responses 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 two architectures at the request rate of 10 per second is interesting. At first glance, it seems that the first architecture 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 Work

In this paper, a framework was introduced to formally specify and evaluate Software Architectures. SA specification is initially described in UML2.0 which 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 software architectures with a powerful model such as TA, we have suggested a rigorous basis to evaluate (and also verify) functional and non-functional attributes of SA. Furthermore, we extended usual TA model to include performance aspects which could be involved in UML2.0 diagrams. We also proposed a 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 future work, we decide to firstly, promote our performance model to support a wide variety of interactions such as asynchronous, anonymous in distributed environments. Secondly, we are going to enhance our framework to include other non-functional attributes e.g. security; this issue will facilitate simultaneous evaluation of several attributes regarding their conflicting natures.

References

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. 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.
3. 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.
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. Maurice H. ter Beek, Gabriele Lenzini, Marinella Petrocchi, Team Automata for Security– A Survey –,Electronic Notes in Theoretical Computer Science, 128 (2005) 105–119.
7. L. Egidi, M. Petrocchi, Modelling a Secure Agent with Team Automata, The Journal of Electronic Notes in Theoretical Computer Science 142 (2006) 111–127.
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, FSEN07, LNCS 4767,423-431, Springer-Verlag Heidelberg, 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. Federica Aquilani, Simonetta Balsamo , Paola Inverardi, Performance analysis at the software architectural design level, Performance Evaluation 45, Elsevier, (2001) 147–178.
20. M.Sharafi, Developing a Framework to Formal Specification and Evaluation of Software Architectures, Final Report of PhD Thesis, Faculty of Computer Engineering, Azad University of Tehran (Science and research branch of), August 2007.