Computing and Informatics, Vol. 32, 2013, 797–826

PERFORMANCE MODELING AND ANALYSIS OF SOFTWARE ARCHITECTURES SPECIFIED THROUGH GRAPH TRANSFORMATIONS

Mahdi Rahimi Naddaf, Vahid Rafe

Department of Computer Engineering, Faculty of Engineering
Arak University, Arak 38156-8-8349, Iran
e-mail: m-rahiminaddaf@arshad.araku.ac.ir, v-rafe@araku.ac.ir

Communicated by Ulrich Eisenecker

Abstract. Software architecture plays an important role in the success of modern, large and distributed software systems. For many of the software systems – especially safety-critical ones – it is important to specify their architectures using formal modeling notations. In this case, it is possible to assess different functional and non-functional properties on the designed models. Graph Transformation System (GTS) is a formal yet understandable language which is suitable for architectural modeling. Most of the existing works done on architectural modeling and analysis by GTS are concentrated on functional aspects, while for many systems it is crucial to consider non-functional aspects for modeling and analysis at the architectural level. In this paper, we present an approach to performance analysis of software architectures specified through GTS. To do so, we first enrich the existing architectural style – specified through GTS – with performance information. Then, the performance models are generated in PEPA (Performance Evaluation Process Algebra) – a formal language based on the stochastic process algebra – using the enriched GTS models. Finally, we analyze different features like throughput, utilization of different software components, etc. on the generated performance models. All the main concepts are illustrated through a case study.

Keywords: Graph transformation system, PEPA, performance model, software architecture

1 INTRODUCTION

Software architecture plays an important role in the success of modern, large and distributed software systems. In an architecture-centric development approach only principal design decisions are considered at the architectural levels and many unnecessary details are removed. This yields more efficient, more effective and faster development of software products through software architecture [1]. Considering system designs at the architectural level, especially when they are specified accurately by means of formal methods, might considerably decrease the cost, increase the quality of the product and decrease the risk associated with them [2].

The importance of architectural analysis will be more prominent while dealing with some properties of system like performance which yields the timing behavior. Performance optimization of developed software may lead to architecture redesign which will bring cost and time overflow. Thus, performance prediction of a software system is necessary in the early development stage when the architecture is designed.

Specifying the architecture is a major requirement for achieving the aforementioned goals. To specify an architectural model some notations are required with the following properties: *expressiveness* to be understandable especially for domain experts, *formality* to reason on a more abstract level, supporting *dynamic architectures* and finally supporting *refinement* in the architectural level. None of the existing Architecture Description Languages (ADLs) support all of them. Thus Thöne [3] has proposed a combination of UML and graph transformation as a visual, yet formal approach to model (and reason about) component based architectures, according to this observation.

Graphs provide a universally adopted data structure, as well as a model for the topology of component-based systems. At the same time, a variety of visual notations used in computer science (i.e. UML) can be easily seen as graphs and thus graph transformations are involved, either explicitly or behind the scenes, when specifying how these models should be built and interpreted, and how they are refined over the time and mapped to implementations.

In addition, as we will explain in the subsequent sections, GTS has simple primitives and familiar concepts and is flexible for defining some profiles in computer domain and especially can provide behavioral modeling facilities in a wide area of abstraction.

However, modeling per se is not enough; it must be complemented with proper approaches to assess the quality of the designed models. In this paper, we present an approach to performance modeling and analysis of GTS specifications. To do this, we use the component-based style presented in [3] for modeling component-based software architectures using GTS. We enrich the component-based style with the performance information (e.g. the rate of behavior mechanisms and component operations). Then, we generate the performance models from the enriched GTS specifications in terms of PEPA [4, 5].

PEPA (Performance Evaluation Process Algebra) is a formal language, based on stochastic process algebra, to specify and analyze performance models. PEPA offers

several attractive features which are not available in previous performance modeling paradigms. The most important of these are:

compositionality, the ability to model a system as the interaction of subsystems,

formality, giving a precise meaning to all terms in the language, and

abstraction, the ability to build up complex models from detailed components, disregarding the details when it is appropriate to do so.

Queuing networks offer compositionality but not formality; stochastic extensions of Petri nets offer formality but not compositionality; they offer abstraction mechanisms neither. In contrast, PEPA can model components interactions and their behaviors. Also, PEPA primitives are very close to architectural concepts. For example, component is a building block of PEPA models and this is a major concept in software architecture.

Compositionality is explicit in PEPA and is provided by combinators. This is useful for model construction especially when models are constructed systematically, by either elaboration or refinement and this is a major requirement in the architectural modeling.

After transformation and generating the PEPA model, we analyze different performance features like throughput, utilization of different software components on the generated PEPA model. Finally, the results are considered as a feedback to improve the designed architecture. Figure 1 shows the proposed framework.

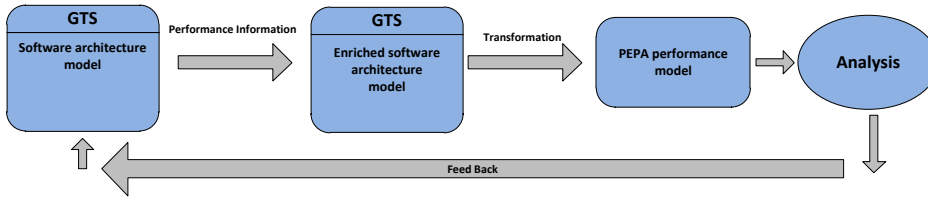

Figure 1. The proposed framework to performance modeling and analysis

The rest of the paper is organized as follows. Section 2 surveys the related approaches. In Section 3, we briefly introduce the required background, i.e. software performance engineering, graph transformation system, the core component-based style modeled through GTS and PEPA. In Section 4, we explain our proposed approach to enrich the style with the performance information. Section 5 presents our used approach to generate performance model (i.e. PEPA model) from the enriched style. In Section 6, we show the analysis approach along with our experimental results. Finally, Section 7 concludes the paper and evaluates our approach.

1.1 Running Example

To illustrate our proposal, we use a business application *"e-Map Router"* presented in [6]. This is a simple example that we use in this paper to present and exemplify our approach.

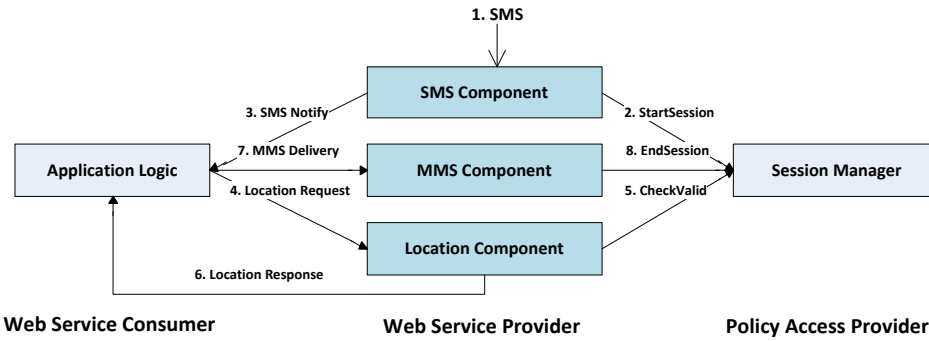

Figure 2. The main components of the e-Map Router system

The running example is an application that finds the nearest restaurant for a user and displays the results on the user's mobile phone. This application displays only one place for each request. Figure 2 depicts the main components of the system (at abstract level). *Web Service Consumer* component encompasses application logic; *Web Service Provider* includes three independent components; *MMS* and *SMS* components are responsible for SMS and MMS communications, respectively; *Location* component performs searching services for received requests; and finally, *Policy Access Provider* controls the security mechanisms considered.

The numbers on the arrows of Figure 2 show the behavior of the system as an informal scenario. In this scenario the client activates a service by sending a SMS. Consequently, the *Session Manager* starts a session. Then, a notification is sent to the application. The application requests the client's location from a location web service. The web service contacts the session manager to control the validity of the request (*checkValid*). *checkValid* is the security processing center of our case study. So, those received requests enter to this point and the security consideration of the system for responding this request will be reviewed. It is valuable to measure the impact of their processing time on the overall performance of the system. Then, the validity check is performed by the *Policy Access Provider*. If it succeeds, the location web service finds the nearest location and passes its map as a MMS to the *MMS Component*. *MMS Component* delivers the result to the user and finally terminates the session.

2 RELATED WORKS

Performance engineering is the main subject of this research. Performance engineering can be used for performance analysis of a lot of products, i.e. process models and software systems.

For process model context, Fritzsche et al. [7, 8], proposed to apply model driven performance engineering. They basically took process models, enriched them via a staged process using model transformations to finally execute a simulation for performance analysis which covers non-functional properties.

Our research is more concentrated on performance modelling and analysis of software architectures. There are several researches which have a similar process depicted in Figure 1. Differences between these researches are around the selected notations and the method of conversion between them. Some of these researches focus on architecture of general systems while the others try to concentrate on special architectures like SOA. Formal notations have been used for specifying architecture in some of them, while others have used informal notations. Some of them have tools with feedback support while others may be applied manually.

Pooley [9] uses UML for modelling the architecture. He models architecture with regard to performance using combination of sequence and collaboration diagrams and a special combination of collaboration and state diagrams. Performance model in this research is specified through PEPA notation and is generated manually.

Mitton and Holton [10] have a research like Pooley [9] but they use only UML state diagrams for architectural modelling.

Petriu and Shen [11] have used UML for architectural modelling but performance model is based on LQN. Transformation of UML model to LQN has been done with graph transformation system. Balsamo and Marzolla [12] have also followed the referred process. They have applied annotated use case, activity and collaboration diagrams for architectural modelling. In this research, performance model is based on multi class queuing networks.

Andrea D'Ambrogio [13] has developed a tool that the referred process can be followed with automatically. It can be used to architectural design and will increase the productivity of the designer. In this tool, architecture is designed using UML and a LQN model is generated for performance.

[14] is about a tool which uses UML for architectural modelling and provides some features for injecting performance information to architectural model. The research emphasises the point that the analysis result should be in the form of UML; this way designers can improve their models.

[15] is a Ph. D. thesis; introduces pa-UML, a modelling language based on UML which is extended for performance. In this research, SPN is the notation used for performance modelling.

The aforementioned researches use UML for architectural modelling. UML is familiar and understandable for domain experts. UML is semi-formal and formal analysis is not feasible directly through it. Dynamic architectures cannot be described through UML. UML extending is a long term activity. With UML, refine-

ment consideration, which is necessary in architectural modelling, cannot be handled directly.

Garlan and Spitznagel [16] have provided an approach for performance modelling and analysis of some architectural style. AESOP [17] has been used for modelling architecture and performance modelling is done through LQN. Some tools have been provided for transforming architectural model to LQN model. AESOP has special provisions for styles. This notation has not offered some features for analysis and has not a formal semantics. Hence this is similar to UML.

Another research which is about special purpose architectures is [18]. This research proposes a framework for performance analysis of SOA based systems. In this research BPE4W, which is a language for specifying web services, is transformed to SPN models. Because of using BPE4W, this approach is applicable for service oriented architecture domain and cannot be extended to other styles. BPE4W can also be used for specifying these architectures and no analysis is supported through it, while in our approach we use GTS for architectural modelling which is capable of formal functional and non-functional analysis

[19] has proposed a kernel language named KLAPPER to face both the heterogeneous design level notations for component-based systems, and the variety analysis methodologies for performance in component-based architectures. KLAPPER captures the relevant information for the analysis of non-functional attributes of component-based systems, with a focus on performance. Using this kernel language, a bridge between design-oriented and analysis-oriented notations is established. Therefore defining a variety of direct transformations from the former to the latter will reduce to the less complex problem of defining transformations to/from the kernel language. This work is not an approach for performance analysis of software architectures. It proposes less effort for generating the performance models through architecture specifications. It is worth to point out that transformation causes losing some information in the source model and KLAPPER suggests several transformations.

[20] has proposed PCM (Palladio Component Model) to specify component-based software architectures in a parametric way. This model considers factors affecting the perceived performance of a software component like influences by external services, changing resource environments, or different input parameters. It has developed a tool capable of simulating instances of the PCM to obtain performance metrics. This approach does not follow the framework of Figure 1 and does not use the familiar and common performance modeling notations and their tools. It is necessary to prove the correctness of the results in its simulation engine and formal functional analysis of software architecture has not been considered, while our approach uses the proved performance analysis methods and other formal functional analysis is feasible through it.

Heckel et al. [21, 22] introduce the notion of stochastic graph transformation systems (SGTS) to formalize, measure, and predict properties such as reliability. In the SGTS, each rule is associated with its application rate. Continuous Stochastic Logic (CSL) is used to specify reliability properties and verify them through model check-

ing. They have constructed an experimental tool chain consisting of GROOVE [23] for generating the labeled transition system, and PRISM [24] for probabilistic model checking. An adapter connects both tools by translating the transition graph generated by GROOVE into a PRISM transition system specification, incorporating the transition rates as specified in a separate file. Later on, Torrini and Heckel [25], use a bit modification to the idea presented in [21, 22]. They suggest using simulation to do the performance analysis to avoid the state space explosion problem. In fact, in the new work they do not generate all the labeled transition system to avoid the state space explosion problem.

This approach has added the static application rate to the rules while we have the dynamic application rate so that modeler can specify different rates for different instances of rules.

Varró et al. [26, 27] extend the meta-model presented in [3] with the required parameters for reliable messaging in services. Then, they incorporate fault-tolerant algorithms into appropriate reconfiguration mechanisms for modeling reliable message delivery by graph transformation rules. In order to assess the cost of fault tolerance techniques in terms of performance, they transform the extended models to PEPA using VIATRA2 framework [28]. In fact, performance analysis is done considering different parameters and message delivery semantics in reliable message passing for service-oriented architectures. Varró et al. [26, 27] use the same modeling style which we have used and also have the same performance modeling notation, but the performance is considered only for reliable messaging while our approach handles the performance more generally.

A couple of other approaches we can mention are those about functional analysis of GTS specifications (e.g. [29, 30, 31, 32]). Some of them use graph-based algorithms to model check GTS specifications (e.g. GROOVE [23]), while some other approaches make a transformation from GTS to the input language of different model checkers (e.g. CheckVML [30]). In all of these approaches there is nothing related to analysis of non-functional aspects. Instead, it is possible to verify different functional aspects in terms of safety, deadlock freeness, reachability and liveness properties.

3 BACKGROUND

This section introduces the required background for the proposed approach, i.e. software performance engineering, graph transformation system, the core component-based style presented in [3] along with a brief description of PEPA.

3.1 Software Performance Engineering(SPE)

Software Performance Engineering is describing, analyzing and optimizing the dynamic time dependent behavior of a system. Methods and tools have been introduced to provide performance engineering and guarantee that an implemented software system will satisfy its performance requirements. Performance engineering

method and tools should be applied during the whole development process of a software system starting from the early phase. In the 1990s Connie U. Smith proposed a systematic approach called Software Performance Engineering (SPE) method [33] for constructing a software system to meet performance goals from the early phase of software development.

The approach has been inspired in many researchers to create their own methods and tools for performance engineering purposes. On the other hand, Software Performance Engineering (SPE) represents the entire collection of software engineering activities and related analyses used throughout the software development cycle, which are directed to achieving performance goals.

Two general approaches exist related to the SPE. The most common approach is purely *measurement-based*; it applies testing, diagnosis and tuning, when the system under development can be run and measured. The *model-based approach* (see also [34] for a survey of modeling approaches), creates performance models in the early phases and uses quantitative results from these models to tune the architecture and design with the purpose of satisfying performance requirements. Our idea in this paper is in the model-based category.

The common properties which are assessed in SPE are response time, throughput, queuing delays, and utilization of different software and hardware components. Some definitions are as follows:

Response time is broadly defined as the time interval between a user's request for service and the services returning the results.

Throughput is a measure of the number of transactions that receive service over some predefined period of time. For the transaction systems, this would be measured as transactions per second, or TPS.

Utilization of a resource is a measure of how busy the resource is. It is computed as the fraction of the time that the resource is busy by servicing clients divided by the entire time period:

$$\text{Utilization} = \text{time busy}/(\text{time busy} + \text{time Idle}) \tag{1}$$

3.2 Graph Transformation System(GTS)

In this subsection, we briefly describe graph transformation, as a modeling notation which is related to graphs. GTS has some vocabulary which we define through subsequent definitions. These are Type Graph, Host Graph, Graph Rules and Rule Firing.

Briefly, Type Graph is a meta-model which defines the elements of our models, their relations and constraints. This is like the concept of Class Diagram in UML. Host graph is an object diagram for Type graph with relevance to UML class diagrams. The state of our model is depicted through the Host Graph. Host Graph shows the current elements of our model and their relationship.

Graph Rules lead the change conditions in the Host Graph or in the elements of our model. Graph Rules have pre and post conditions. These conditions are the constraints in the host graph. When a pre-condition of a rule in the host graph is satisfied this rule can be fired. When fired, graph elements in pre condition will be changed according to the post-condition.

Definition 1 (Type graph transformation)**.** A type graph transformation system is a triple: $TGS = (TG, HG, R)$, where TG is the type graph, HG is the host graph over TG and R is the set of rules.

Definition 2 (Type graph)**.** A type graph is a tuple $TG = (TG_N, TG_E, src, tar)$ where TG_N is the set of node types, TG_E is the set of edge types and src, tar are functions from TG_E to TG_N. These functions assign to each edge a source and a target node.

Definition 3 (Host graph)**.** A host graph HG, also called instance graph over TG, is a graph equipped with a graph morphism $type_G : HG \to TG$. This assigns a type to every node and edge in HG.

Definition 4 (Graph rules)**.** A graph transformation rule P over a type graph TG is given by $P : L \to R$ where L and R are some graphs over TG. L defines the precondition (LHS) and R is the post condition of the rule (RHS).

The application of a rule to a host graph H replaces a matching of the L in H by an image of R. This is performed by (1) finding a matching of the L in H, (2) deleting a part of the host graph (that can be mapped to L but not to R) producing the context model, and (3) connecting the context model with a matching of the R by adding new nodes and edges (that can be mapped to the R but not to the L) and resulting in a new model H'.

By recursively applying all enabled graph transformation rules to the host graph, a transition system can be generated. Transition systems are frequently used to represent the behavior semantics of software systems. In the case of graph transition systems, one considers graphs as representations of system states. If the resulting state space of the graph transition system is finite, we can easily check different properties (e.g., reachability, safety, and liveness), even for unrestricted forms of graph transformation systems, by searching the state space. For more information of GTS interested readers can refer to [35, 36].

3.3 Core GTS Component-Based Style (CBS)

For modeling architectural styles using GTS, its concepts, i.e. type graph, instance graph and transformation rule should be applied. [3] has proposed an approach to model architectural styles through GTS which we use for modeling our case study. Due to its role in this paper, we describe it briefly in this subsection.

Structural part of each style includes some vocabulary with relational constraints. Component-based style vocabulary includes *Component, ComponentType,*

Port, PortType, Connector, Interface etc. which are modeled through a graph schema (Figure 3).

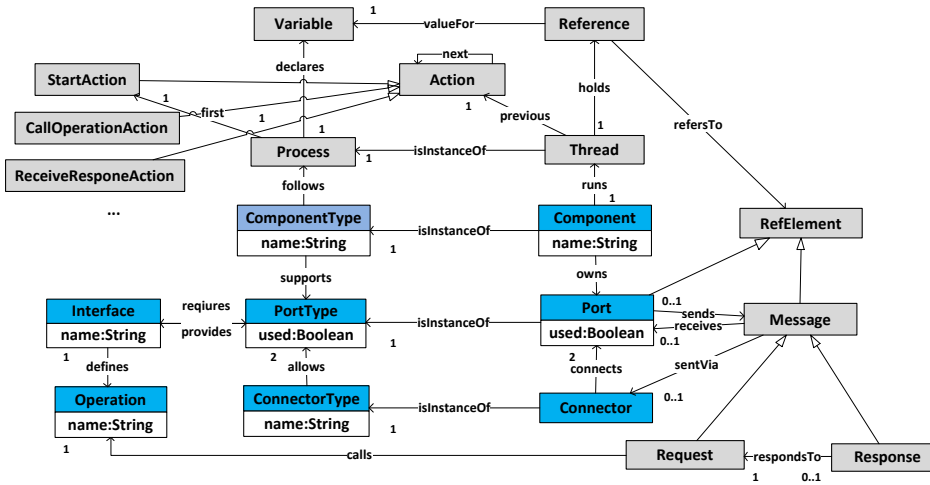

Figure 3. Graph schema of the component-based style

For example, each *Port* is an instance of the *PortType* which introduces the interfaces provided by a component or the interfaces required by it. *ConnectorType* specifies the *PortTypes* which can be the endpoints of a *Connector*. As a constraint, each *ComponentType* has only one *Process* and also a *Connector* connects only two *Ports*.

To specify the architecture of a real system, an instance graph of this graph schema should be instantiated. Figure 4 is the structural part of the architecture of our case study. This graph represents the participant components, their interfaces, operations, static behaviors etc.

In addition to the nodes required for structural modeling, some other nodes are required for behavior modeling. These nodes are shown darker in the graph schema.

Behavior in this architectural style includes some sequence of connection and commutation mechanisms. These mechanisms are modeled through graph transformation rules like connecting two components, sending a request, receiving the response of a request etc. A transformation rule is provided for each of these mechanisms in the model. Some of these rules are *connect*, *openPort*, *callOperation*, *sendResponse*, *receiveResponse*, etc. Firing these rules on the host graph implements the behavior of the architectural style.

Therefore *Message, Response and Request* nodes in graph schema represent the different messages which are passed between components.

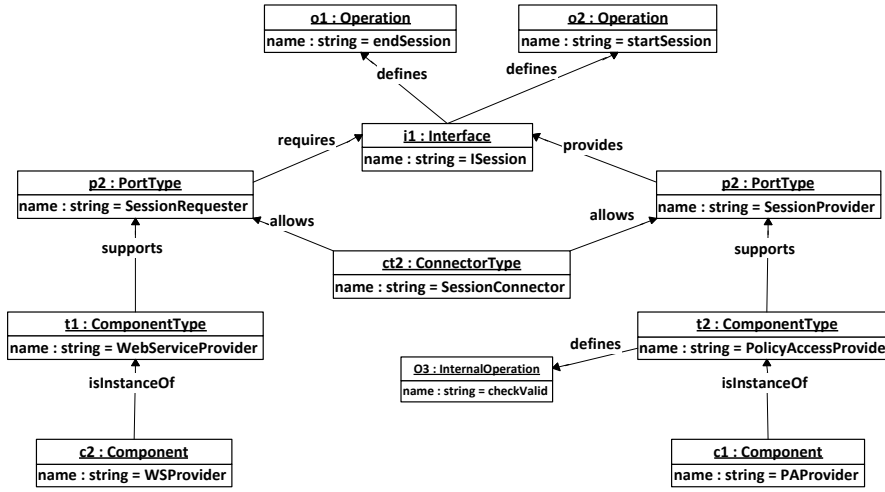

Figure 4. An excerpt of the host graph of our case study

A number of nodes in the graph schema like *Action* and *Process* have been provided to address the control flow which is necessary for managing the sequence of executing the behavioral mechanisms. *Action* is the parent class of all the behavioral mechanisms. Each behavior mechanism such as *callOperation* is an *Action* node in the graph schema. *Process* is a sequence of different *Actions* which specify the behavior of a *ComponentType*. *Thread* is an instance of the *Process* type which specifies the behavior of a *component* instance.

Actions are executed with variable or constant parameters. For example, *OpenPort* action has a constant parameter which defines its port name. Some nodes like *RefElement*, *Reference* and *Variable* are used for modeling parameter passing.

To denote the constant parameter of the action, an edge is established between the Action node and the parameter node with a label starting with #.

Variable parameters are more complex sine these are different for an *Action* in each *Thread*. Each action which has a variable parameter should have an edge to a *Variable* node. For modeling the value of this parameter *Reference* node can be used. This node refers to *Thread*, *Variable* and its value. Therefore in each *Thread* the value of this variable can be distinguished.

Figure 5 shows a rule named *callOperation* which uses all of the aforementioned behavioral modeling elements.

This transformation rule is fired when there is a *component(c)* in the host graph which has established a connection to another *component* and that *component* has an *interface* defining an *operation* named *o*. The *c* component has a *thread* which points at an *Action* node as the previous invoked action. Previous action has a *next* edge to a *CallOperationAction* node as the next action. *CallOperationAction* defines

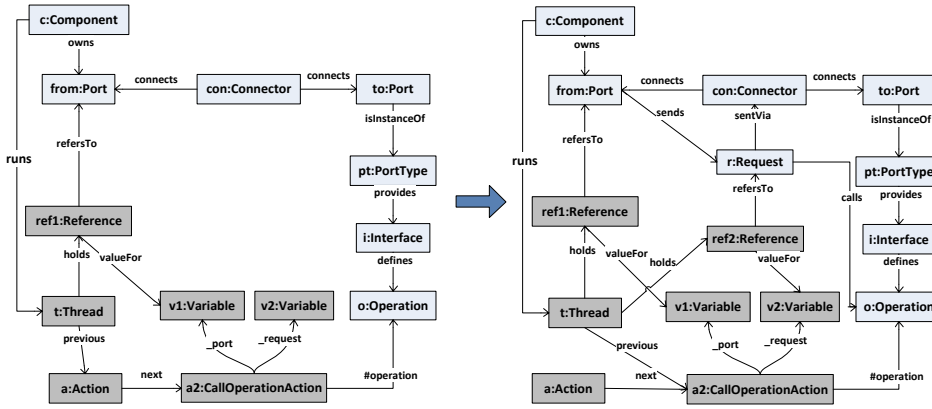

Figure 5. *callOperation* rule [3]

its Operation through an edge labeled *#operation*. Active *thread* in this rule has a pointer to the connecting *port* through *Reference* and *Variable* nodes. This rule represents that c is ready to call the o operation.

3.4 PEPA(Performance Evaluation Process Algebra)

We use PEPA modeling language which is based on Stochastic Process Algebra (SPA) for performance modeling and its toolkit for analysis. PEPA has some key characteristics for performance analysis in the architectural level which makes it suitable for our purpose.

In PEPA a system is described as a set of components and their interactions. In fact, components are the processing elements while activities represent the operations associated with them. Each activity has an *action* type (or simply *type*). Every activity in PEPA has a duration which is a random variable with an exponential distribution. This parameter is called the *activity rate* (or simply *rate*) of the activity; it may be any positive real number, or the distinguished symbol T, which should be read as *unspecified* [4, 5].

An action is represented as a pair (α, r), where α is the type of the action and r (a real number) is the parameter of the negative exponential distribution governing its duration. Whenever a component performs an action, an instance of a given probability distribution is sampled: the resulting number specifies how long it will take to complete the action.

Components and activities are the primitives of the PEPA language; the language also provides a small set of combinators. With these combinators more useful and applicable clauses can be constructed.

Consider a system which is composed from a *WebService*(WS) component and an *Application*($Appl$) component. WS receives request from an $Appl$, processes the request, and sends a response back to the $Appl$. The nature of the WS behavior is

Title	Form	Description
Prefix	$(\alpha, r).P$	Sequential behavior representation
Cooperation	$P < \alpha, \beta > Q$	Cooperating two component(P, Q) for some activities α, β
Choice	$P + Q$	Parallel processing in two components(P, Q)

Table 1. PEPA combinators

sequential. Hence, the *Prefix* combinator should be used. Consequently, its PEPA clause is:

$$WS = (\text{request}, \mathsf{T}).(\text{serve}, \mathsf{r}).(\text{respond}, \mathsf{T}).WS$$

This clause says that WS receives a request with the unspecified rate (T). Then it serves the request with the rate r and finally responds it. This process will be continued in WS.

Appl has two different behaviors, for some cases local processing is done while for the other cases it sends a request to the WS. On the other hand it thinks first, after that for some cases processes its request locally and for other cases sends its request to the WS. Then *Appl* has some choices in its behavior and Choice operator should be used. Its PEPA clause is as follows:

$$Appl = (\text{think}, r1).(\text{local}, m).Appl + (\text{think}, r2).(\text{request}, q).(\text{respond}, p).Appl.$$

The total system behavior is composed of the interaction among these two components (i.e. WS and *Appl*). Thus, the total system equation uses Cooperation combinator and its PEPA clause is as follows:

$$Sys = (Appl \langle \text{request}, \text{serve}, \text{response} \rangle WS).$$

This example has been used as another case study and we will refer to it in the evaluation section. Interested readers can refer to [5] for more information about formal operational semantics of PEPA.

4 STYLE EXTENSIONS FOR PERFORMANCE ANALYSIS

To do the performance analysis on the architectural models specified through GTS, at first we should enrich the style. So, we extend the component-based style presented in [3] by adding performance related information on the system. Thus, we should equip the type graph using the required nodes, attributes and new graph rules to support performance concepts along with modifying some of the existing graph rules. This extension can be considered in the following parts:

- Extending the style for calling internal operations inside the components
- Extending the style for setting the timing information of component operations and behavior mechanisms modeled by rules.

The major existing behavioral mechanisms in the style are communication mechanisms. However, we need to have a new mechanism for modeling some internal processing inside the component. For example, with the existing mechanisms we can model calling the external actions of a component which are reachable through its interfaces. However, for performance modeling there is a serious need to consider some major internal processing in a component. In our case study *checkValid* is an internal operation for *PAProvider* component which is not accessible for other components. This is invoked according to calling other interface operations. We need to inspect some performance concerns about it but existing mechanisms cannot handle it.

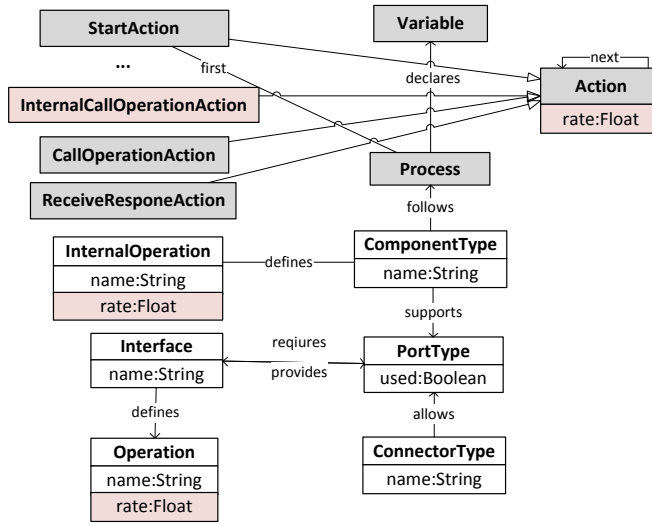

Figure 6. An excerpt of the extended graph schema (modified nodes are colored)

To satisfy this requirement, the structural part of the style (the type graph of Figure 3) should be extended to support a new node named *InternalOperation*. There is a new operation defining in the *ComponentType* which cannot be provided by an *interface*. It can only be called by its associated component privately. So, the behavioral part of the style should be extended to support a new mechanism named *InternalCallOperation* which is used for calling internal operations and *IternalCallOperationAction* is the associated action node for it. Figure 6 shows the extended type graph while Figure 7 depicts the rule *InternalCallOperation*.

To consider the timing information for component operations, their application rates (the rate of the exponential distribution governing the delay of its application) as a new attribute called rate should be added to the related node types. Figure 6 shows the enriched type graph. As it is shown in *InternalOperation*, *Operation* and *Action* are behavior nodes of the schema which have a new float attribute called *rate*.

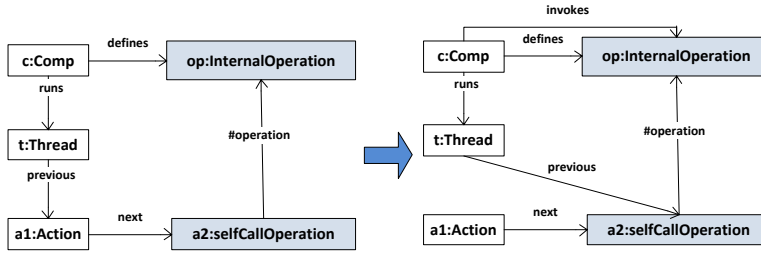

Figure 7. New behavior rule (*InternalCallOperation*)

For the e-Map Router example, the initial configuration of the system can be described through a host graph conforming to the extended type graph of Figure 3. Figure 8 shows an excerpt of the host graph for our case study.

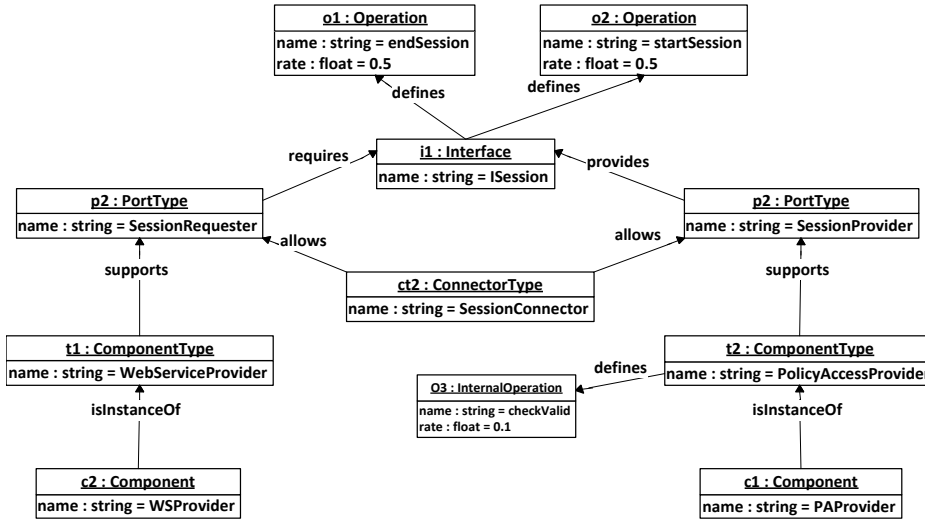

Figure 8. An excerpt of the extended host graph of e-Map Router case study

5 TRANSFORMATION ALGORITHMS

After enriching the GTS models with the necessary performance information, we should generate the PEPA description of the models. To transform the architectural model to PEPA model an algorithm should be followed. This algorithm has two major steps.

- Encoding the structural elements.
- Encoding the behavior of the model.

The behavior of the model is implemented through rule firing; thus inspecting the transition system of the model can help us in extracting the behavior of the components. A big problem is the state space explosion as encountered in producing the state space of the graph model. To overcome this problem we use a reduced state space. To achieve this state space we change the host graph of the model. The new host graph is the same as the main but includes only one instance of each component. The behavior of the components in the new graph is exactly as the original model, because none of the graph transformation rules in the model are dependent to the number of instances of a component in the LHS and RHS of the rules. So, reducing the instances of a component to one instance does not affect the firing conditions of the rules and this causes the behavior of a component does not change.

These steps are described in the following sections. Before that, it is necessary to define some formal notions.

5.1 Formal Definition of the Transition System

The transition system of a GTS model is a tuple

$$TS = (A, C, S, T, CN, \text{beg}, \text{end}, \text{state}, \text{trans})$$

in which:

- A is the countable set of the corresponding action names of the performance model in architectural level. This set is prepared when identifying the structural elements.
- C is the countable set of component types in the architectural model. This set is prepared when identifying the structural elements.
- S is the set of transition state nodes.
- T is the set of transition edges between states.
- $\text{beg} : T \to S$ is a function. This function denotes the starting node of a transition edge.
- $\text{end} : T \to S$ is a function. This function denotes the ending node of a transition edge.
- CN is the set in the form of $\{(c, n) \,|c \in C, n \in \mathbb{N}\}$ which denotes the last state index for each component in C. Initial setting is zero indexes for each component. Also this set is used for each node of the transition graph.
- $\text{state} : S \to CN \bigcup \{\emptyset\}$ is a function. This function denotes to each node of the transition system a set of pairs representing the component state index in it.

- trans : $T \rightarrow (A, \mathbb{R}, C, C \bigcup \{\emptyset\})$ is a function. This function associates each edge of the transition graph to some information. This information is the related action name of an edge (A), the application rate of its action ($\mathbb{R}$), the active component or the component which fires this action (C), and finally the cooperative component in this action or the component which is passive according to this action (C), it may be empty. How to prepare this information will be described later while identifying the structural elements.

5.2 Formal Definition of the PEPA Model

A PEPA model is a tuple

$$PM = (A, C, PE, CS)$$

where A is the set of action types and C is the set of components.

- $PE = \{(P, i, j, \alpha, r) \mid P \in C, i, j \in \mathbb{N}, r \in \mathbb{R}, \alpha \in A\}$ is the set of PEPA prefix clauses of the model.
- $CS = \{(c_1, c_2, A_1) \mid c_1, c_2 in C, A_1 \subseteq A, c_1 \neq c_2\}$ holds the cooperation information of the components. Each element of this set represents the cooperation of two components on an action set.

5.3 Encoding the Structural Elements

According to the formal definition of the graph, in this step, the elements of the C, A and the resulting information of *trans* function of TS should be calculated.

The elements of C can be extracted from the host graph. The nodes which are instances of *ComponentType* in the host graph are the components of the model. For example, in the host graph of our case study (Figure 4) there are one component of the *PolicyAccessProvider* type and one instance of the *WebServiceProvider* type.

Extracting the elements of A can be done using the transition system. The generated transition system in GROOVE has some information for the edges as represented in Figure 9. This information includes the related rule name and the corresponding node identifiers in the host graph. Accordingly, we can name the associated activity of this edge to generate the elements of A. The mechanism of naming is different for each transformation rule.

For example, if the related rule of the edge is *callOperation*, the sender component, the receiver component, the operation name and the duration rate of this operation can be distinguished using the transition edge information and the host graph. These can be used for TS edge information (in the form of $(A, \mathbb{R}, C, C \bigcup \{\emptyset\})$.

For *callOperation* edges, if the sender of this request is *caller_comp*, receiver is *callee_comp*, the requested operation is *op* on the *callee_comp* defined interfaces and the duration of this action for *caller_comp* is r then the activity name of this edge which should be added to the set A can be a combination like *caller_comp_call_callee_comp_op*. This combination guarantees the uniqueness of the actions name in the model.

Furthermore, the existing edges in the TS need the cooperation between components of their related activity. The cooperation in this style is related to some sort of behavioral mechanisms. These are *CallOperation/ ReceiveCall, SendResponse/ ReceiveResponse, Connect and Disconnect*. This means that *ReceiveCall* can be coordinated with regard to a *CallOperation* edge, then for *ReceiveCall* edges the caller component of *CallOperation* should be distinguished according to the edge information, the host graph and the LHS/RHS of the rule.

5.4 Encoding the Behavior of the Model

This step is divided into three sub-steps. In the first step, the prefix and the choice behavior should be detected, in the second step the cooperation cases should be distinguished and finally, in the third step the PEPA model clauses should be generated which can be used in the PEPA toolkit.

It is necessary to point out that different behavior states of a component is represented through several state numbers in its PEPA model. Also, the nodes in the transition system denote the behavioral states of the model. Therefore, we have considered the state indices of each component in the nodes of the transition system. The initial setting for the nodes is empty and only the first node will have indices equal to zero for each component as the initial value.

The transformation algorithm saves the current state number of each component in the CS set. The algorithm uses it for controlling the traversing of the transition system. Traversing can be done according to DFS or BFS algorithms.

5.4.1 Prefix and Choice Behavior Transformation

For transformation of the transition system to PEPA model, the component set (C) and the action set (A) are handled in the same way and for generating PE and CS a set of rules on the transition system should be applied. These rules are described in the following.

Rule #1: Prefix behavior:

$$\begin{aligned}
&\forall t \in T, \\
&\text{trans}(t) = (\alpha, r, P, Q), \\
&S_1 = beg(t) \neq \emptyset, \\
&S_2 = end(t), (P, n) \in CN \\
&\Rightarrow \\
&PE = PE \bigcup (P, n, n+1, \alpha, r) \\
&S_2 = S_1 + (P, n+1) - (P, n) \\
&CN = CN - (P, n) + (P, n+1).
\end{aligned}$$

This rule generates the associated PEPA prefix clause of an edge. Also, it moves the state information of components from the starting node of the edge to its ending node. Figure 9 depicts the usage of this rule.

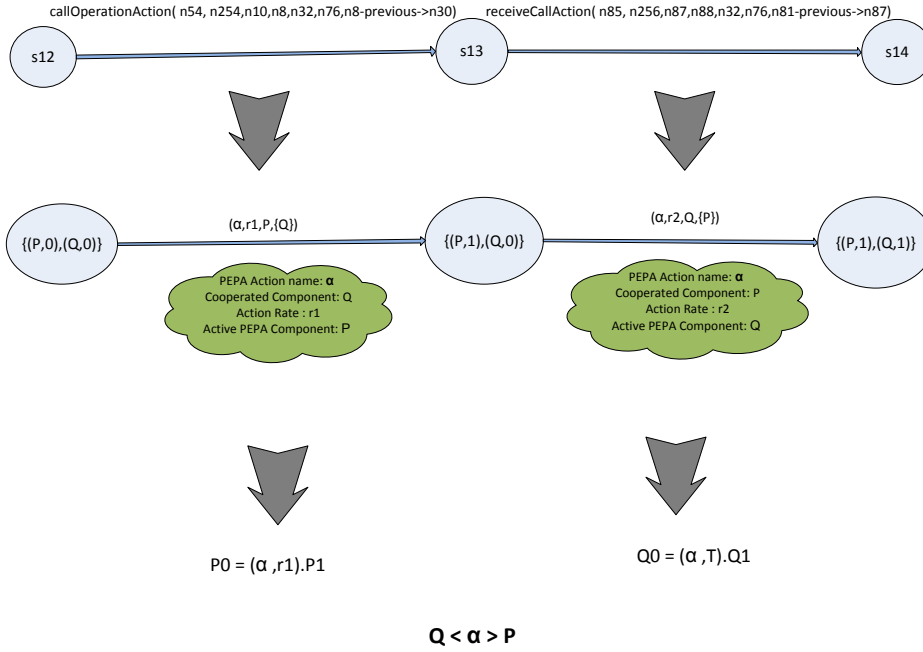

Figure 9. prefix behavior transformation rule

Rule #2: Cooperation behavior:

$$\forall t \in T, \text{trans}(t) = (\alpha, r, P, Q \neq \emptyset), \Rightarrow CS = CS + (P, \alpha, Q).$$

These rules will generate the PEPA prefix and choice clauses and achieve the coordination set of our model. According to these steps, the PEPA model of our components will be ready. Now it is time to deal with the cooperation clauses of our model. These clauses represent the cooperation of components to mimic the behavior of the system.

5.4.2 Cooperation Behavior Transformation

To achieve the cooperation clauses, a new undirected graph will be assumed (CG):

$$CG = (V, E, copcls).$$

This graph has a set of nodes (V) and a set of edges (E). Nodes include the name of a component from C or some cooperation clauses like $c_i < A_1 > c_j$ where $A_1 \subseteq A$

and $c_i, c_j \in C$. The cooperation clause of each node is specified through the *copcls* function. This function maps each node to a number of alphabetic strings which denotes the cooperation clause of the node. The first clause in each node is the name of a component.

Edges of this graph represent the cooperation between components on some action types. For example, if there is a cooperation between $c_i, c_j \in C$ on action type set A_1, then an edge exists between two nodes v_i and v_j such that $copcls(v_i) = c_i$ and $copcls(v_j) = c_j$ and the edge is labeled as A_1.

This graph should be constructed applying the following rule on the CS set which is prepared in the prior sections. According to this rule, for each cooperation of two components in CS one edge should be established between two components in CG.

$$\begin{aligned} \forall cs_1 &= (c_1, c_2, \alpha) \in CS \Rightarrow \\ V &= V + v_2 + v_1 \\ copcls(v_1) &= c_1, copcls(v_2) = c_2 \\ E &= E + v_1 \overset{\alpha}{\rightarrow} v_2 \end{aligned}$$

This graph should be transformed to some disjoined nodes according to the following rule. Resulting nodes include the cooperation clauses suitable for this system.

$$\forall (v_1, v_2, A_1) \in E \wedge v_1 \neq v_2$$

$$\Rightarrow$$

$$\begin{aligned} V &= V - v_1 - v_2 \\ V &= V + v_3 \\ copcls(v_3) &= (v_1 < A_1 > v_2) \end{aligned}$$

$$\forall e = \left\{ v_i \overset{A_2}{\leftrightarrow} v_j \right\}, (v_i = v_1 \vee v_i = v_2) \Rightarrow E = E - e + \left\{ v_3 \overset{A_2 \cup A_1}{\leftrightarrow} v_j \right\}$$

According to this rule, two related nodes will be merged to one node and the new node will be replaced by the edges of the two old nodes. The new node will have the coordination clauses of these two nodes. Figure 10 depicts the application of this rule.

5.5 PEPA Model Clause Generation

After generating the PEPA model elements, some processing should be done to prepare the text of the PEPA model. We can use some auxiliary rule to generate this.

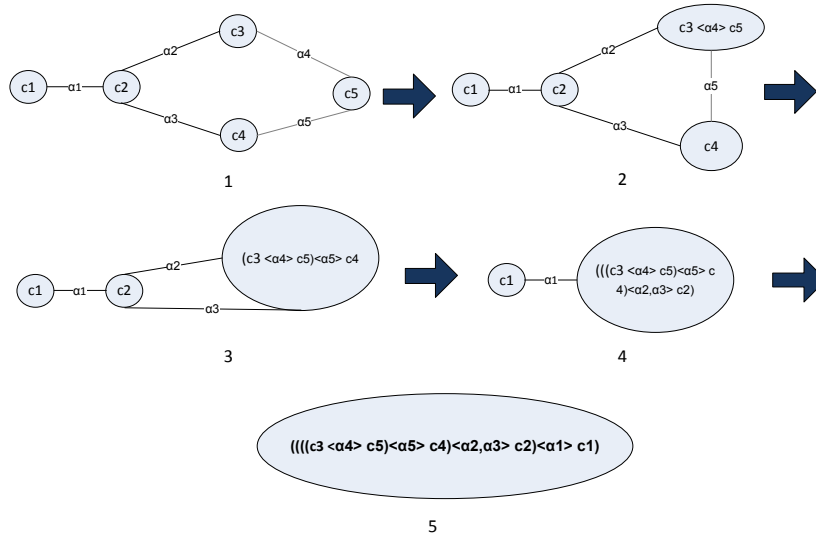

Figure 10. Cooperation clause generation

Rule #1: Prefix clause extraction

$$\begin{aligned} \forall pe_i \in PE \text{ in } PM & \\ pe_i &= (P, n, m, \alpha, r_1) \\ \neg\exists pe_j \in PE \text{ in } PM & \\ pe_j &= (P, n, k, \beta, r_2) \\ &\Rightarrow \\ \text{Clause} &= \text{Clause} + \text{“}P_n = (\alpha, r_1).P_m\text{”} \end{aligned}$$

Rule #2: Choice clause extraction

$$\begin{aligned} \forall pe_i \in PE \text{ in } PM & \\ pe_i &= (P, n, m_i, \alpha_i, r_i), i = 1 \ldots \\ &\Rightarrow \\ \text{Clause} &= \text{Clause} + \text{“}P_n = \ldots + (\alpha_i, r_i).P_{m_i} + \ldots\text{”} \end{aligned}$$

Rule #3: Cooperation clause extraction

$$\begin{aligned} \forall v \in V \text{ in } CG & \\ \mathit{Clause} &= \mathit{Clause} + \mathit{copcls}(v) \end{aligned}$$

6 PERFORMANCE ANALYSIS

In this section we explain how different non-functional properties can be analyzed on the generated PEPA model.

Some of the considerable performance parameters for each system are the efficient number of different components, so the throughput is an important issue. Also, the designers may be interested to know the effects of some components on the other components. Furthermore, the designers are interested to know the utilization of different component states, the response time of significant system activities, etc. The result of analysis will help designers to make a better decision before implementation phase. For example, for our case study it is interesting to know the throughput of the deliverMMS component since it is an important criterion for customer satisfaction and shows the performance of the system.

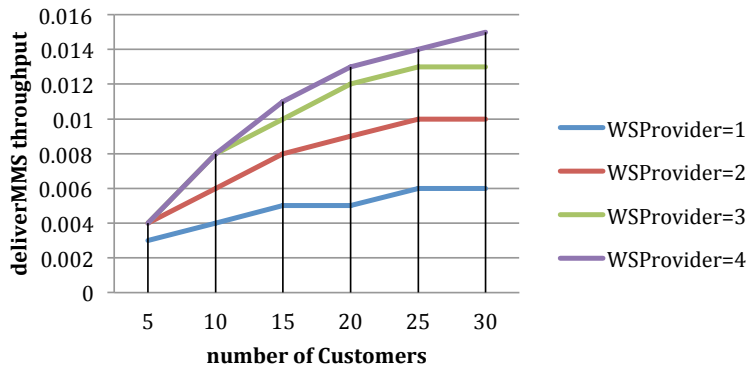

Figure 11. Throughput of the deliverMMS for different numbers of customers and WSProvider

Figure 11 depicts the throughput of the deliverMMS for different numbers of clients and WSProvider components. The parameters used in this analysis are shown in appendix B.

As the figure shows, increasing the number of WSProvider components when there are a few customers has no effect, while for nearly 30 customers in the system it has significant effect. Choosing three instances of WSProvider in the system along with 30 customers, we can conclude that two WSConsumer is enough in the same way as for WSProvider.

Now with these decisions, we want to do a sensitivity analysis on the checkValid operation processing time. In other words, we want to know whether a more secure but slower algorithm for this operation is more suitable or a less secure approach which is faster. Also, we want to find the proper numbers of PAProviders.

Such effects are measured in Figure 12 where the deliverMMS throughput is plotted against do_checkvalid for different PAProvider pool sizes.

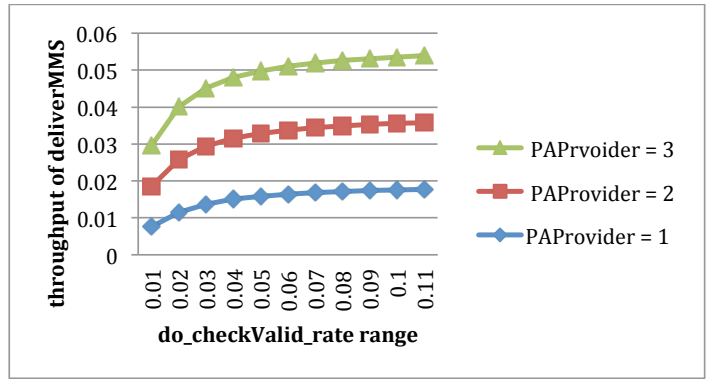

Figure 12. Throughput of the deliverMMS

As shown in Figure 12, by increasing the rate of the check_valid operation to a specific point, the throughput of the deliverMMS will be increased and after that point there is no considerable improvement. So, we can conclude that the rates before the slope point are proper choices for check_valid operation. For our case study, as it is shown, having two components is more efficient than having one or three components.

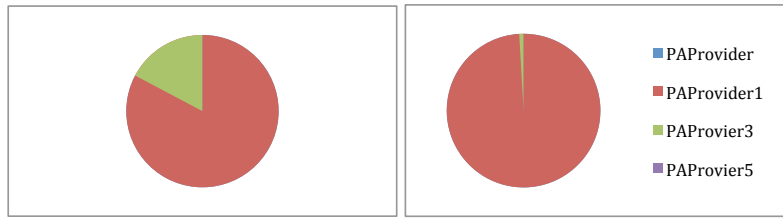

Figure 13. Utilization of PAProvider states (two instances)

Utilization of component states that can be derived from PEPA toolkit is also a metric for finding the suitable number of a component. Figure 13 and Figure 14 depict the utilization of PAProvider component for two and three pool sizes.

Since PAProvider in its first state (named PAProvider) is waiting for request, zero utilization for an instance of component shows that no request was sent to it. Then, as Figure 14 shows, using the third instance of PAProvider has no effect in the overall system performance improvement.

7 CONCLUSION AND EVALUATION

In this paper, we propose an approach to performance analysis of software architectures specified through GTS styles. To do so, we enrich GTS models with the

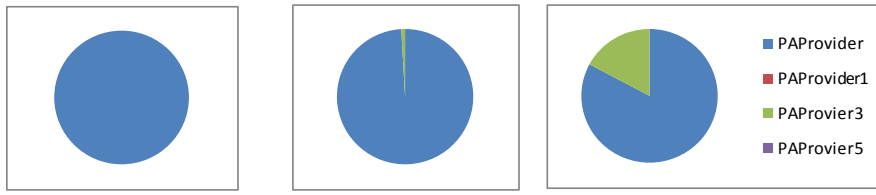

Figure 14. Utilization of PAProvider states (three instances)

necessary performance information. Then, we generate the PEPA model from the enriched GTS model. At the end, we show how different non-functional properties can be analyzed on the generated performance models.

GTS has a good potential for considering architectural refinement. All efforts to model and analysis software in the architectural level are not sufficient without considering refinement with regard to the temporal nature of the architecture. Our selected notations in this paper have the potential to be refined. Thus, this approach can be applied to performance modeling of the other levels of the refined architecture.

PEPA which is based on SPA is the performance modeling notation in our approach. PEPA primitives are very close to the existing concepts in architectural modeling; and it has some attractive features like compositionality, formality and abstraction which have been described in the previous sections.

We have proposed an approach which has several applications. GTS architectural model can be analyzed qualitatively through GROOVE and other GTS based tools for functional properties. The PEPA model of the architecture can be used for quantitative analysis of some functional properties through PRISM probabilistic model checker. Finally, performance properties can be evaluated through a PEPA model in the PEPA toolkit. The result of applying the last case has been described in the performance analysis section.

The presented approach has also some drawbacks. GTS is not popular and familiar in software modeling. Its tools are not mature and a big effort is needed to expand these tools. For example, the host graph of our case study has more than 200 nodes, so managing this host graph was a big grief for us during this research.

State space explosion is an old and familiar obstacle for large models like our case study. As shown in Table 2, we could not extract the state space for some cases. Our transformation algorithm is based on state space and then our approach does not work for some large models, even with diminishing the host graph for having smaller state spaces.

Using host graph for transforming the architectural model to PEPA model is a raw idea which can be considered as a future work.

Our transformation algorithm can be developed with a programming language. In addition, it can be applied with some graph transformation tools like VIATRA [28] because the source and the target models can be seen as graph and a set of rules

can be established for transformation. However, our approach can be faster since it uses a simple edge traversing algorithm like DFS while VIATRA needs pattern matching algorithms on a huge state space graph. According to Table 2 applying rules in a huge state space will not be feasible in many models; but transforming the two graph models with graph transformation rules would help us to prove the correctness and completeness of our work.

	Case study	# Components	# Host Graph Nodes	# Rules
1.	e-Map router	4	218	11
2.		6	222	11
3.		8	230	11
4.	Client/server[1]	2	43	11
5.		3	46	11
6.		4	49	11
7.		5	52	11
8.		6	55	11

	# State Space Nodes	# State space Transitions	Time (ms)	Memory(KB)
1.	2 573	7 113	6 263	19 979
2.	18 412	64 820	131 123	230 620
3.	overflow	overflow	–	–
4.	29	46	133	1 626
5.	234	593	519	2 934
6.	1 775	6 200	2 654	26 041
7.	overflow	overflow	–	–
8.	overflow	overflow	–	–

Table 2. Exprimental Results

To prove the correctness of our approach, we have used a case study as a benchmark which is used in the PEPA tutorial [6] and we got the same performance analysis results.

The GTS style used in our approach does not support the modeling mechanisms like loop, resource handling, switch, etc. explicitly; but in our idea, stochastic nature of performance modeling language offers us these facilities implicitly. For example, looping of an action for an indefinite time can be seen as a new action with a suitable duration rate which simulates the probability of executing the first action in the loop. Thus for performance prediction in these cases, this pattern seems to be adequate and this solution can be used for other mechanisms.

Acknowledgment

We would like to thank Prof. Jane Hillston (currently at the Edinburgh University), for her supports in this research especially in the utilization of PEPA and its toolkit.

REFERENCES

[1] Taylor, R. N.—Medvidovic, N.—Dashofy, E. M.: Software Architecture: Foundations, Theory, and Practice. John Wiley 2008.

[2] Baresi, L.—Heckel, R.—Thöne, S.—Varró, D.: Style-Based Modeling and Refinement of Service-Oriented Architectures. Software and Systems Modeling, Vol. 5, 2006, No. 2, pp. 187–207.

[3] Thöne, S.: Dynamic Software Architectures: A Style-Based Modeling and Refinement Technique with Graph Transformations. Ph. D. thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn 2005.

[4] Hillston, J.: Tuning Systems: From Composition to Performance. The Computer Journal, Vol. 48, 2005, No. 4, pp. 385–400.

[5] Hillston, J.: A Compositional Approach to Performance Modelling. Ph. D. thesis, Department of Computer Science, University of Edinburgh CST-107-94, April 1994.

[6] Clark, A.—Gilmore, S.—Hillston, J.—Tribastone, M.: Stochastic Process Algebras. In Proc. of the 7th International Conference on Formal Methods for Performance Evaluation, 2007, pp. 132–179.

[7] Fritzsche, M.—Picht, M.—Gilani, W.—Spence, I.—Brown, J.—Kilpatrick, P.: Extending BPM Environments of Your Choice with Performance Related Decision Support. BPM 2009, pp. 97–112.

[8] Fritzsche, M.—Gilani, W.: Model Transformation Chains and Model Management for End-to-End Performance Decision Support. GTTSE 2009, pp. 345–363.

[9] Pooley, R.: Using UML to Derive Stochastic Process Algebra Models. Private communication, in Proceedings of the 15th UK Performance Engineering Workshop, 1999, pp. 23–33.

[10] Mitton, P.—Holton, R.: PEPA Performability Modeling Using UML Statecharts. Proceedings of the 16th Annual UK Performance EngineeringWorkshop 2000, pp. 19–33.

[11] Petriu, D. C.—Shen, H.: Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In Proceedings of the Computer Performance Evaluation/TOOLS, 2002, pp. 159–177.

[12] Balsamo, S.—Marzolla, M.: Performance Evaluation of UML Software Architectures with Multiclass Queuing Network Models. In: Proc. of the 5th International Workshop on Software and Performance (WOSP 2005), ACM Press 2005, pp. 37–42.

[13] D'Ambrogio, A.: A Model Transformation Framework for the Automated Building of Performance Models from UML Models. In Proc. of 5th International Workshop on Software and Performance (WOSP '05), New York, NY, USA 2005, pp. 75–86.

[14] Canevet, C.—Gilmore, S.—Hillston, J.—Prowse, M.—Stevens, P.: Performance Modeling with UML and Stochastic Process Algebras. IEE Proceedings: Computers and Digital Techniques, Vol. 150, 2003, No. 2, pp. 107–120.

[15] Merseguer, J.—Campos, J.: Software Performance Modeling Using UML and Petri Nets. LNCS, Vol. 2965, Springer 2004, pp. 265–289.

[16] Spitznagel, B.—Garlan, D.: Architecture-Based Performance Analysis. In Proc. of the Software Engineering and Knowledge Engineering (SEKE '98), 1998.

[17] Garlan, D.—Allen, A.—Ockerbloom, J.: Exploiting Style in Architectural Design Environments. Proceedings of SIGSOFT '94 Symposium on the Foundations of Software Engineering, December 1994.

[18] Holanda, H. J. A.—Barroso,G. C.—Serra, A. B.: SOASPE: A Framework for the Performance Analysis of Service Oriented Software. SBSI, 2009, pp. 204–215.

[19] Grassi, V.—Mirandola, R.—Sabetta, A.: Filling the Gap between Design and Performance/Reliability Models of Component-Based Systems: A Model-Driven Approach. Journal of Systems and Software, Vol. 80, 2007, No. 4, pp. 528–558.

[20] Becker, S.—Koziolek, H.—Reussner, R.: Model-Based Performance Prediction with the Palladio Component Model. In Proc. of 6th International Workshop on Software and Performance (WOSP '07), 2007, pp. 56–67.

[21] Heckel, R.—Lajios, G.—Menge, S.: Stochastic Graph Transformation. In Proc. of the Second International Conference on Graph Transformation, Springer 2004.

[22] Heckel, R.: Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks. In Proc. of ICTAC '05, LNCS, Vol. 3722, Springer 2005, pp. 53–69.

[23] Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In Proc. of AGTIVE03, LNCS, Vol. 3062, Springer 2004.

[24] Kwiatkowska, M.—Norman, G.—Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In Proc. of TOOLS '02, LNCS, Vol. 2324, Springer 2002, pp. 200–204.

[25] Torrini, P.—Heckel, R.—, Ráth, I.: Stochastic Simulation of Graph Transformation Systems. In FASE 10, 2010, pp. 154–157.

[26] Gönczy, L.—Kovács, M.—Varró, D.: Modeling and Verification of Reliable Messaging by Graph Transformation Systems. In: Proc. of theWorkshop on Graph Transformation for Verification and Concurrency, Elsevier 2006.

[27] Gönczy, L.—Déri, Z.—Varró, D.: Model Transformations for Performability Analysis of Service Configurations. Springer-Verlag 2009, pp. 153–166.

[28] Varró, D.—Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 2007, Vol. 68, No. 3, pp. 214–234.

[29] Rafe, V.—Rahmani, A. T.—Baresi, L.—Spoletini, P.: Towards Automated Verification of Layered Graph Transformation Specifications. IET Software 2009, Vol. 3, No. 4, pp. 276–291.

[30] Schmidt, A.—Varró, D.: CheckVML: A Tool for Model Checking Visual Modeling Languages. In Proc. of UML 2003, LNCS, Vol. 2863, Springer 2003, pp. 92–95.

[31] Baldan, P.—Corradini, A.—König, B.: Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. In Proc. of CONCUR04, LNCS, Vol. 3170, 2004, pp. 83–98.

[32] Baresi, L.—Rafe, V.—Rahmani, A. T.—Spoletini, P.: An Efficient Solution for Model Checking Graph Transformation Systems. Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 213, Elsevier Science B. V. 2008, pp. 3–21.

[33] Smith, C. U.: Performance Engineering of Software Systems. Addison-Wesley 1990.

[34] Balsamo, S.—Marco, A. D.—Inverardi, P.—Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Transactions on Software Engineering, Vol. 30, 2004, pp. 295–310.

[35] Baresi, L.—Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Proc. of the ICGT 2002, Springer 2002, pp. 402–429.

[36] Ehrig, H.—Ehrig, K.—Prange, U.—Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science – An EATCS Series). Springer 2006.

8 APPENDICES

8.1 Appendix A (The generated PEPA code for our case study)

```
sendSMS_rate = 0.0010;
startSession_rate = 0.5;
endSession_rate = 0.5;
startSession_resp_rate = 1 000 001;//rapid
notify_rate = 0.1;
getLocation_rate = 0.1;
do_checkValid_rate = 0.1;
checkValid_resp_rate = 100 000;
checkValid_rate = 100 000; // intermediate operations should be very fast
deliver404_rate = 100 000;
getLocation_resp_rate = 99.0 * 100 000;
createMap_rate = 0.05;
sendMMS_rate = 0.2;
deliverMMS_rate = 0.02;
Client = (sendSMS,sendSMS_rate).Client1;
Client1 = (deliver404, infty).Client + (deliverMMS, infty).Client;
WSProvider = (sendSMS,infty).WSProvider1;
WSProvider1 = (startSession,startSession_rate).WSProvider2;
WSProvider2 = (startSession_resp, infty).WSProvider3;
WSProvider3 = (notify, notify_rate).WSProvider4;
WSProvider4 = (getLocation,infty).WSProvider5;
WSProvider5 = (checkValid,checkValid_rate).WSProvider6;
WSProvider6 = (checkValid_resp,infty).WSProvider7;
WSProvider7 = (deliver404,deliver404_rate).WSProvider12
    + (getLocation_resp,getLocation_resp_rate).WSProvider10;
WSProvider10 = (sendMMS,infty).WSProvider11;
WSProvider11 = (deliverMMS,deliverMMS_rate).WSProvider12;
WSProvider12 = (endSession,endSession_rate).WSProvider;
PAProvider =  (startSession,infty).PAProvider1 + (checkValid,infty).PAProvider3
    + (endSession,infty).PAProvider;
PAProvider1 = (startSession_resp,startSession_resp_rate).PAProvider;
PAProvider3 = (do_checkValid,do_checkValid_rate).PAProvider5;
PAProvider5 = (checkValid_resp,checkValid_resp_rate).PAProvider;
WSConsumer = (notify,infty).WSConsumer1;
```

WSConsumer1 = (getLocation,getLocation_rate).WSConsumer2;
WSConsumer2 = (getLocation_resp, infty).WSConsumer3
+ (deliver404,infty).WSConsumer;
WSConsumer3 = (do_generateMap, createMap_rate).WSConsumer4;
WSConsumer4 = (sendMMS,sendMMS_rate).WSConsumer;
Client[30] ⟨sendSMS, deliverMMS, deliver404⟩ WSProvider[2]
⟨startSession, startSession_resp, endSession, checkValid, checkValid_resp⟩
PAProvider[2] ⟨deliver404,notify,getLocation,getLocation_resp,sendMMS⟩
WSConsumer[2]

8.2 Appendix B (Parameters used for the performance analysis of the case study)

parameter	value	description
sendSMS_rate	0.001	rate at which customers request service
startSession_rate	0.5	rate at which a session can be started
endSession_rate	0.5	rate at which EndSession can be done
startSession_resp_rate	1 000 000	rate at which startSession response can be sent. a big value shows negligible effect of this operation in system
notify_rate	0.1	notification exchange between client and provider
do_checkValid_rate	0.05	rate at which checkValid is done in PA-Provider
checkValid_resp_rate	10 000	rate of checkValid response sending
checkValid_rate	10 000	rate of checkValid request preparing and sending
deliver404_rate	100 000	rate at which WSProvider acts when checkValid response is false
getLocation_resp_rate	99.0 ∗ 100 000	rate at which WSProvider acts when checkValid response is true, in this case map should be generated. The rate of this case is 99 times more than false response for checkValid
createMap_rate	0.05	rate at which map is created
sendMMS_rate	0.2	rate at which MMS messages can be sent via the Web Service
deliverMMS_rate	0.02	rate at which MMS messages can be sent from provider to customer

Mahdi Rahimi Naddaf received his B. Sc. from Sharif University and his M. Sc. from Arak University, both in software engineering. His research interests include software methodology, software architecture, software security and formal methods. He works in AFAGH software company as a project architect and manager.

Vahid Rafe received his B. Sc., M. Sc. and Ph. D. in software engineering from Iran University of Science and Technology. He was also visiting researcher at Politechnico di Milano (Italy). He is currently an Assistant Professor at Arak University. His research interests include formal specification and verification of software systems, model transformation and refinement and software architectures.