

## Modeling of component diagrams using petri nets

Sima Emadi<sup>1</sup> and Fereidoon Shams<sup>2</sup>

<sup>1</sup>Computer Engineering Department, Islamic Azad University, Maybod Branch, Yazd, Iran

<sup>2</sup>Computer Engineering Department, Shahid Beheshti University, Tehran, Iran

Emadi@maybodiau.ac.ir, f\_shams@sbu.ac.ir

### Abstract

With the growing use of UML diagrams for software architecture description and the importance of non-functional requirements evaluation at software architecture level, filling the scientific gap between architect and requirement analyst is considered to be significant. Software architects are not usually familiar with non-functional requirement analysis and are not able to analyze such requirements easily. On the other hand, non-functional requirements cannot be evaluated directly by UML diagrams. Therefore, the architect should annotate additional information of the non-functional requirements to software architecture description and then an executable model can be produced. These executable models can be petri nets, queuing networks, stochastic process algebra and etc. One kind of the UML diagrams that can be used to describe software architecture is component diagram. In this paper, we propose a new algorithm that enables an architect to transform a component diagram into an executable model based on different extensions of petri nets. Moreover, we show how to use this petri net model for performance evaluation and simulation and the implications of this transformation are described completely. Finally, to represent the usage of our proposed algorithm, we consider a case study as an example.

**Keywords:** Component diagram, non-functional requirements, executable model, petri nets.

### Introduction

Nowadays, one of the most noticeable tasks of an architect is the production of executable model from software architecture description for non-functional requirement evaluation. The use of software architectures for predicting quality attributes of the overall system is one of the original motivations in the field of software architecture (Bass *et al.*, 2003; Clements *et al.*, 2002). Especially the quality of an architectural design has a great influence on achieving non-functional requirements of the system such as performance, fault-tolerance, security, reliability and so on. If an architect adopted the architecture that was not appropriate for the requirements, the system of low quality would be produced and it would lead to wasting huge amount of cost and development time because he or she has to redevelop the system of a certain quality. To avoid wasting cost and time resulting from the choice of inappropriate architectures, describing an architectural design formally and validating its appropriateness at the architectural design step, before starting the implementation of the system is very helpful (Bass *et al.*, 2003; Clements *et al.*, 2002). To evaluate non-functional requirements, first an architect should describe software architecture with ADL, MSC or UML (OMG, 2003) among which UML is considered to be the most important to describe the software architecture. Then the analyzer must evaluate the non-functional requirements by using an executable model. This kind of evaluation is difficult, time-consuming and expensive.

Different approaches for performance evaluation in software architecture level have been listed in (Balsamo & Simeoni, 2001), where the authors have compared some recently proposed approaches to the transformation of software architecture described by UML diagrams into performance evaluation models. They have considered different types of UML diagrams and derived different types of performance models, such as queuing networks, stochastic petri nets, stochastic process algebras and simulation models. Bernardi *et al.* (2002) have proposed the automatic translation of state charts and sequence diagrams into generalized stochastic petri nets, as well as a composition of the resulting net models suitable for reaching a given analysis goal (Bernardi *et al.*, 2002). Andol *et al.* (2000) have represented an approach to derive automatically a performance evaluation model based on queuing network from a software architecture specification described by MSC. Cortellessa and Mirandola (2000) have shown how a software system modeled by sequence, deployment and use case diagrams can be translated into a performance model based on queuing network.

Most of the researches have used a queuing model. The quality attributes considered by them are restricted to measuring performance and are also for communication-based architectures. In addition, queuing networks are not suitable for qualitative attributes but very useful in stochastic analysis of every kind of architecture. A queuing model with probability factors should be developed, but the applicability of this model for different types of non-functional requirements as well as

performance quality is restricted. That is, they can work well when we apply them to specific non-functional quality attributes and specific domains. When we apply them to other quality attributes such as reliability, we should redesign the queuing model to a new one.

Furthermore, many researchers have proposed approaches to requirement evaluation of component based software development (Gomaa *et al.*, 2005; Kounev, 2006; Silva *et al.*, 2007). Most of these researches have not evaluated architecture for decreasing the knowledge gap between the design and requirement analysis phases. To evaluate non-functional requirements at software architecture level, the architect encounters lack of knowledge to map an informal space (e.g. software architecture described by UML) to a formal space (e.g. executable model based on petri nets). Bastide and Barboni (2006) proposed a formal model of components, and in particular provided a formal semantics framework for the main concepts of software components. This paper describes the mapping between the component model and coloured petri nets, the chosen behavioural notation. For each component of the assembly, the paper provides a formal behavioural specification. Finally, it describes the denotational semantics that allows to automatically construct a single, unstructured signal net from the behaviours of all the components and their interconnections. This net describes the behaviour of the assembly as a whole.

None of these researches have utilized component diagram for non-functional requirements evaluation of software architecture based on petri nets as well. On the other hand, in an executable model based on petri nets, when we add a quality attribute to be measured, we just attach the values denoting the attribute to tokens and adopt expressions for calculating the quality attribute value from the newly attached values on the tokens (Fukuzawa & Saeki, 2002). The petri net models are very suitable for qualitative and quantitative analysis purposes. Moreover, petri nets can be used for software architecture description as well as software architecture analysis. As it is clear, each executable model is suitable for evaluating of some non-functional requirements. In addition, some of non-functional requirements may conflict and it is not suitable to evaluate them concurrently in the software architecture level. Petri nets model does not completely support the analysis of all kinds of non-functional requirements. For example a petri net based model that is suitable for performance evaluation of the software architecture may not be necessarily acceptable for security evaluation of the architecture.

In our previous works (Emadi & Shams, 2008; Emadi *et al.*, 2008), we proposed a framework to obtain an executable model based on petri nets. To do this, additional information of non-functional requirements should be annotated to the software architecture descriptions, and then such descriptions must be

transformed into general syntax based on different extensions of petri nets. According to this framework, the input of our algorithm is a software architecture described by a set of use case diagrams, sequence diagrams and component-diagrams. Use case diagrams model the functions of system components, sequence diagrams model the interactions between system components (i.e., the messages exchanged between the components) and component diagrams model the structure of components and interfaces between them.

In this paper, we propose an algorithm for the transformation of a component diagram to an executable general model based on different extensions of petri nets. This algorithm decreases the knowledge gap between the software architecture design and requirement analysis. It must be mentioned that after obtaining the executable model, for each nonfunctional requirement, we must indicate usability of the diagram and we must annotate the diagram with non-functional requirement input parameters. Then we must transform these input parameters to tokens of places or guards for arcs and transitions in petri net model.

In this paper, to analyze the software performance, from the description of the software architecture enriched by additional information a target petri net model is generated. After that, the target model must be evaluated and measures for the defined figures must be calculated. The designers decide whether and how the software architecture should be refined from the analysis of the results of the evaluation step. If the designers do not experience any performance problems after the interpretation of the performance results, they proceed to develop the software system. Otherwise, to meet the performance requirements, they modify the software architecture and re-iterate the performance analysis process over the new software architecture. In the software architecture modification the designers will consider the insights gained from the performance analysis.

### Component diagrams

As discussed in the previous section, the proposed methodology uses component diagrams to model static structure of the system at the software architecture level. A component diagram in UML 2.0 is defined by a set of components connected to each other by their interfaces. An interface defines a set of component operations. The interfaces of components can be classified as provided interfaces and required interfaces. For each component, some tasks in the form of operations are assigned. Some of these operations are used by the component alone, but some need the operations of other components or they fulfill the needs of other components. These operations are described by defined interfaces between components (OMG, 2003). In UML 2.0, a component can have two different views, external view and internal view. The external view exhibits only the publicly visible properties

and operations which are encapsulated in the provided and required interfaces. The wiring between components is specified by dependencies or connectors between component interfaces. The internal view shows the component internals that realize the functionality of the component. An external view is mapped to an internal view by using dependencies which are usually shown on structure diagrams. In our work we consider both of the external and the internal views of the components.

*The role of component diagram in performance evaluation*

In order to analyze the software performance, the UML design has to be annotated with performance characteristics, i.e., performance input parameters. The UML-SPT profile gives the syntax to define them through the use of predefined stereotypes and attributes (OMG, 2005). We use the component diagram to specify the structure of the software system and we annotate it with service rates for the provided interface and the scheduling policies of each software components. The annotated component diagram defines the service centers and information on the parameterization of them, such as the type of the service center (e.g. servers with waiting queue or a delay), the rates of the services they provide and the scheduling policies they use to extract jobs from their waiting queues. The set of processing jobs of component service center is defined by component interfaces. Since each component can have a considerable number of interfaces with other components, therefore the relevant components may have several service classes, one for each interface. In addition, the needed time for services accomplishment is added to each interface. At the moment, for simplicity, we assume that an interface contains just an operation from which it inherits the name. This assumption eases the annotation of the service demands in component diagram.

**The transformation of a component diagram to petri net**

As mentioned before, the aim of this paper is to fill the gap between software architecture design and analysis of non-functional requirements. Therefore, a model must be placed between these two spaces (architecture & analysis spaces) to assist the architect in evaluating the process. In this paper, the architecture space is described by use case, sequence and component diagrams and the analysis space by petri nets. Therefore, an algorithm should be designed to map the component diagrams to petri nets. The component diagram can be transformed to the petri net through the following steps (Emadi & Shams, 2009):

- Refining the components of a component diagram. This refinement is used to constrain the ordering of responsibilities of each component

and collaborator components.

- Transforming each of the refined components independently to a petri net. This transformation will map each component and its responsibilities to equivalent petri net.
- Combining the obtained petri nets from the previous step based on the interfaces between components. In this step from extracted petri nets of the previous step, a final petri net can be obtained. This petri net is based on collaborator components and interfaces between components.
- Defining the initial marking on the resulting petri net.

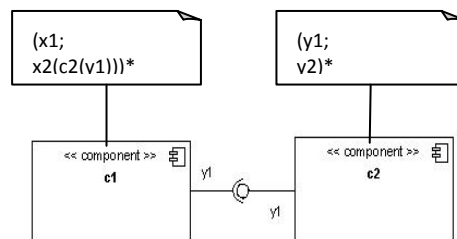
In the rest, we will explain the details of each step.

*Refinement of the components in a component diagram*

It is difficult to characterize the possible behaviour of a component through the informal representations. Therefore, to specify the constraints on a component's behaviour, we use the path expression formalism (Campbell & Habermann, 1974). Path expressions were proposed by Campbell and Habermann as a method for the specification of process synchronization. They provided a clear structured approach to the description of shared data, coordination and the communication between concurrent processes (Campbell & Habermann, 1974; Shams, 1996). Path expressions are used to specify the exact semantics of ordering between the operations within a component. This ordering prescribes the possible behaviour of a component and helps to clarify what should be done in terms of primitive operations such as sequence, alternation, and iteration of operations. The formal syntax of path expressions is derived from regular expressions to represent primitive operations such as sequence, selection, iteration and parallel. In this paper, the introduced path expressions are: ";" that means the execution sequence of operations, "," which means the selection of an operation from a set of operations, "+" that means the one or more iterations, "\*" which indicates the zero or more iterations and "||" that means the parallelism of two operations. The "(" used in path expression extends the effect of the iteration operator on the path inside the brackets. The operations which need other components to perform their tasks or needed by other components are specified as well.

By adding path expressions, we constrain the responsibilities, and these constraints will automatically cover the collaborators field in a component. It means that the collaborator components do not require separate constraints when the activation of the collaborator components totally depends on the activation of the responsibilities. For the sake of simplicity, this ordering should be structured using primitive constructs such as sequence, alternation, iteration and concurrency. If the name of the collaborator component operation is not the same as the

Fig. 1. A. Refined component diagram.

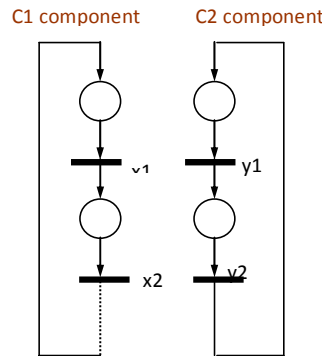


component operations, the name of this operation comes in front of the collaborator components. For example, the Fig. 1 indicates a refined component diagram. As shown, component c1 includes two operations x1 and x2, which are executed sequentially. To execute x2, it needs y1 from component c2.

*Petri net representation of each component*

By adding path expressions to component operations, now we have a precise representation of operations for each component. To accomplish such operations, a component needs to interact with other components; therefore, we need to add the semantics of such interactions. Thus the next step described in the next section defines the properties of the current step as well as the new properties relating to other components. By using the corresponding Petri net representation, we provide a precise semantics of operations for a component (Emadi & Shams, 2009). In this paper, transitions denote the activation of operations within components and the places used to represent pre-

*Fig. 2. Equivalent petri net for each component of fig. 1.*



as shown in Fig. 3.

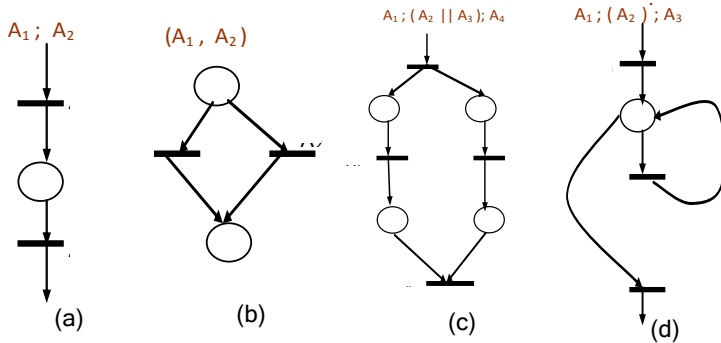
This transformation will enable the architect to simulate and analyze the software architecture. Furthermore, a lower level of abstraction can be reached, as the modeler needs to include more details in the software architecture description when extracting the Petri net representation of the model. This, of course, only covers the semantics of component operations in terms of Petri nets where no interaction has been specified. However, interactions must be considered in such representations. For the sake of simplicity, we specify the semantics of interactions separately. Thus, suitable semantics must be added to complete these representations during the mapping of the collaborators field of a component to Petri net.

*Petri net representation of the component's collaborators*

In previous section, we obtained an equivalent petri net for each component. In this section every interaction will be realized using its corresponding message passing representation in terms of petri net. Interaction semantics implicitly exists in path expressions and one needs only to explicitly represent such semantics. This can be done by providing a Petri net definition of synchronous, asynchronous, and rendezvous types of interactions between two components. To clarify the meaning of each interaction, we must add extra semantics to the previous definitions. In the step described in this section, the transformed component will be combined based on their interfaces. An interface defines a set of operations of the component. In what follows, we use the terms "server component" for the component which provides an operation for the other components, and "client component" for the component which requires an operation from other components (Emadi & Shams, 2009). For example, Fig. 4 shows a complete petri net for

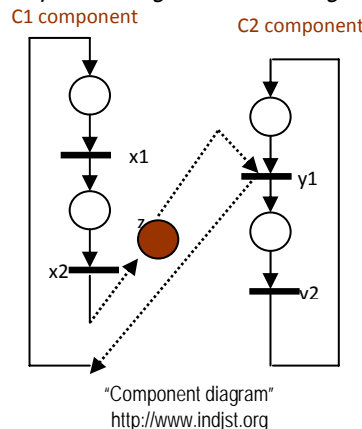
*Fig. 3. Equivalent petri net for primitive operations*

- (a). Sequential path, (b). Selection path,
- (c). Parallel path, (d). Iteration path



conditions to fire a transition. We represent the transitions and places of each component aligned in the same column to indicate their belonging to the same associated task. Path expressions are used to represent the constraints on the firing of such transitions and the constraints on the execution sequence of the operations. It should be mentioned that the operations which need other components are connected to the consecutive operations through dotted lines. Fig. 2 indicates the equivalent petri net for each component of Fig. 1. In this figure, separate petri nets are drawn for components c1 and c2. The component c1 contains two

*Fig. 4. Complete petri net of the component diagram shown in fig. 1.*



the component diagram shown in Fig. 1. This figure shows an example of the interaction representation that provides a precise semantics for interaction. In this transformation, the transition of the client component with the help of a shared place is connected to the transition of the server component and then the output of this transition is connected to the place of the client component. As it can be perceived from Fig. 4, the output of the transition  $x_2$  is connected to the input of the transition  $y_1$ . The place  $z$  as a shared place plays two roles: the output place for the client component and the input place for the server component. In the current approach, only one transition for each call has been used. The returning arc from the corresponding transition in the server component denotes the retrieval of results issued by the service. However, the notion of collaborators when applied to components broadly covers any sort of interaction which can take place between two components.

Moreover, from the synchronization viewpoint, we can classify interactions into two types: 1- Asynchronous: the client component does not need to wait after sending the message to its server components and also can proceed without waiting for the result, 2- Synchronous:

the client component, after sending a message to its server component has to wait until receiving the response.

Fig. 5 shows a typical sequential path expression for the interactions and their equivalent petri net. As mentioned earlier, after obtaining Petri nets for each formalized component, we need to know which operations of the server component will be used to provide the required service. For example in Fig. 5a, when the component  $c_1$  and  $c_2$  are interacting, the  $s$  operation from the  $c_2$  will be used. Fig. 5b shows a selection path and its equivalent petri net. Depending on the selection of the actions  $p$  or  $q$  from component  $c_1$ , the required transition will be activated. Furthermore, a selection path may not necessarily have interactions on both sides. Fig. 5c shows iteration path and its equivalent petri net. In this figure after activating the operation  $s$  from component  $c_2$ , its return point is the place from where the activation of that operation has originated. Fig. 5d shows a concurrent path and its equivalent petri net. The completion of the two concurrent paths is a necessary precondition for activating the action which appears after the concurrent paths. However, in this example, only one interaction was required. In general case, two or more

Fig. 5. (a). Sequential path, (b). Selection path, (c). Iteration path, (d). Concurrent path with interactions & their equivalent petri nets.

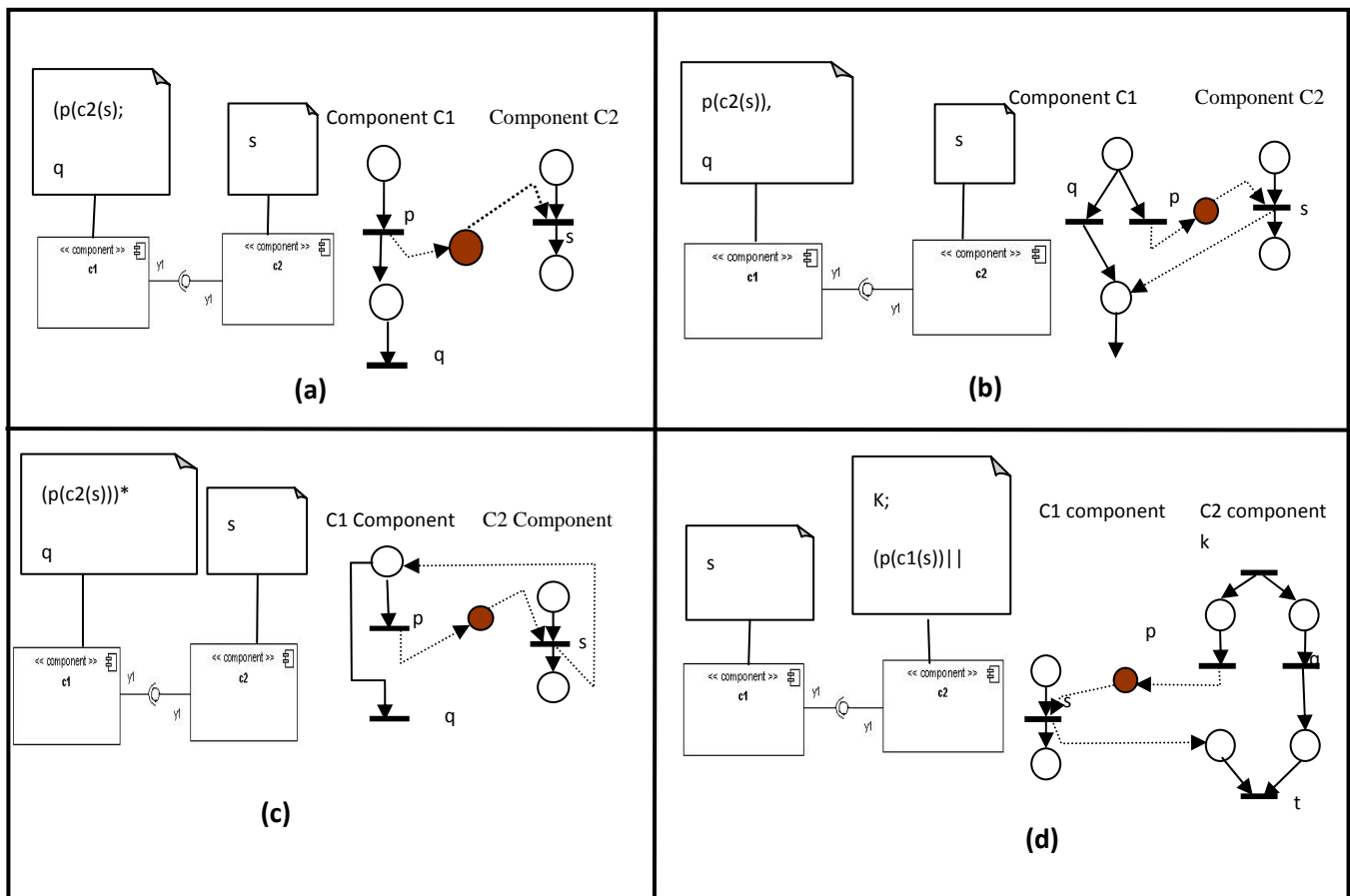
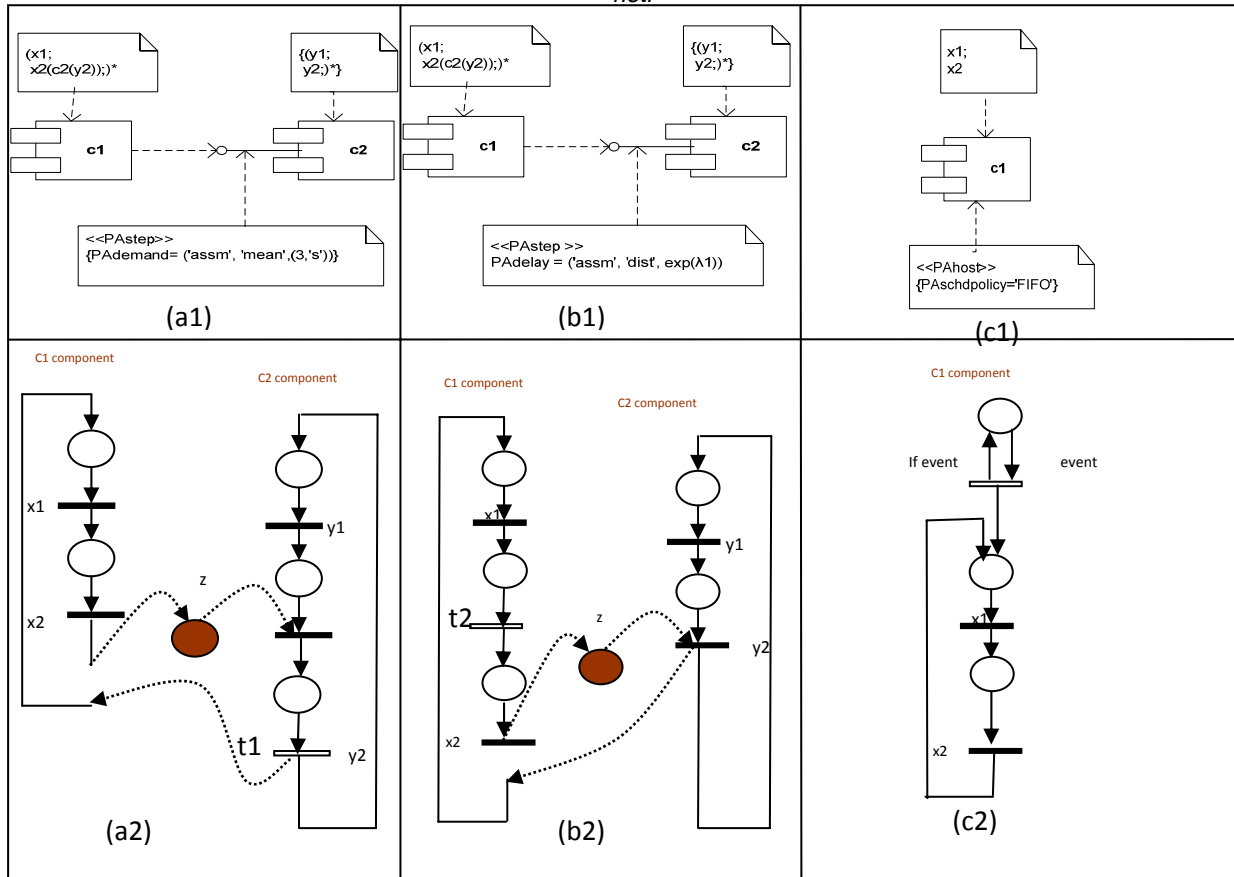


Fig. 6. (a1). Component diagram with PA demand tag value, (a2). Equivalent petri net, (b1). Component diagram with PA delay tag value, (b2). Equivalent petri net, (c1). Component diagram with PA schd policy tag value, (c2). Equivalent petri net.



concurrent interactions are also possible. As specified earlier, the notation of a formalized component should be read from top to bottom and from left to right. To keep this notation clear, in Fig. 5d, the parallel operator of the path has been placed on a different line in its component.

*Petri net representation of the annotated component diagram*

In this section, we propose a new transformation method of component diagram enriched with performance annotations into performance models based on petri net, where the metrics can be effectively computed via stochastic analysis. To carry on the performance analysis we need additional information on the software system generally missing in the software architecture description. Such data are strictly related to the performance aspects and are used in the petri net parameterization and in the workload definition. They are the operational profile of the system (modelling the way the system will be used by the users), the workload entering the system, the service demand required by a request (job) to the system components it visits, and performance characterization of the system components (service rate, scheduling policy, waiting queue capacity). We annotate the UML diagrams with such information by using the UML profile for the schedulability, performance

and time (OMG, 2005). First, we describe the meaning of the performance input parameters.

As mentioned before, the annotated component diagram provides information on the parameterization of the service centers, the rates of the services they provide and their scheduling policies. We use the <<PAhost>> stereotype to annotate the components in the diagram. <<PAhost>> stereotype models an active resource. Also we specify the <<PASchdpolicy>> tag value for logical devices that indicates the scheduling policy of the waiting queue of the components. Since each component can have a considerable number of interfaces with other components, therefore the relevant components may have several service classes, one for each interface. In addition, the needed time for services accomplishment is added to each interface. This information is specified by either <<PAdemand>> or <<PAdelay>> tag values of the <<PAstep>> stereotype. These stereotypes associates to each component interface (Di Marco & Inverardi, 2004). <<PAstep>> models a passive resource step in a performance analysis scenario.

The center is a waiting center if its interfaces are annotated with the PAdemand tag value, otherwise, if PAdelay tag value is used, the center represents a delay center. The PAdemand attribute specifies the duration of

the operations as random variables exponentially distributed. Obviously, the interfaces of a component cannot be annotated by both tag values at the same time. The component diagram enriched by performance annotation and equivalent Petri net is shown in Fig. 6. In this figure, the <<PAdemand>>, <<PAdelay>> and <<PASchdpolicy>> tag values are transformed to timed transitions. The value associated to the tag PADemand and PADelay defines the firing rate of the timed transition. If the operation has <<PAdemand>> tag value (Fig. 6a2), it can mean that the operation needs time, so t1 transition is defined as timed transition.

If the operation has <<PAdelay>> tag value (Fig. 6b2), it can mean that the operation is requested by providing component in delay, so t2 transition is defined as timed transition.

We also specify the <<PASchdPolicy>> tag value (Fig. 6c2) for each component that indicates the scheduling policy of the waiting queue of the components. In equivalent Petri net of these components, assigned time to tokens indicates the scheduling policy.

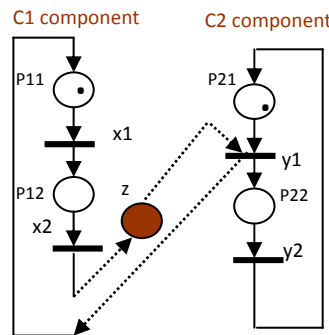
As shown in Fig. 6, by adding the performance parameters to component diagram, only t1 and t2 timed transition and one place are added to the resulting petri net model.

**Simulating the dynamics of the petri net models**

After providing a petri net model of the component diagram, it can be used for simulation. Therefore, a simple enactment of the software architecture description can be achieved. Such a simulation enacts the model to aid understanding, checking, analyzing of the non-functional requirements and improving the software architecture specification. A multiple thread of execution within a component diagram can be simulated by viewing the movement of tokens through different components at the same time and by firing a transition that shows the activation of the responsibility of a component. It must be mentioned that a petri net tool can be used to simulate the resulting model after the initialization of places. The normal initial state of a modeled component diagram will assist in its understanding and improving the expected behaviour of a component. In fact, initialization provides the starting point for presenting the dynamics of the resultant petri net. The places between two columns show the flow of message exchanges. Fig. 7 shows a petri net model of the Fig. 1 after the initialization of places. For example, when the component c1 needs component c2, it must send a message to the component c2. The required message for activating such a service by c2 is named y1. We name the message and the operation of each component the same. The pre-conditions for the activation of the operation y1 are: 1- A token in the place p21 of c2. It means that c2 at this stage has a free slot to

accept an item from the c1. 2- A token in the place z. It means that c1 has requested a free slot in the c2 by sending a message to the buffer. The returned arc from the c2 to c1 only denotes the thread of control. It means that the c1 (client) can't proceed until it receives the expected result from the c2 (server). Using petri nets also clarifies the behaviour of the components, because they provide a powerful graphical representation of such behaviours. Simulating the behaviour of the components by means of a petri nets tool is a further advantage of providing a petri net representation for component diagrams.

Fig. 7. Petri net model of the fig. 1.



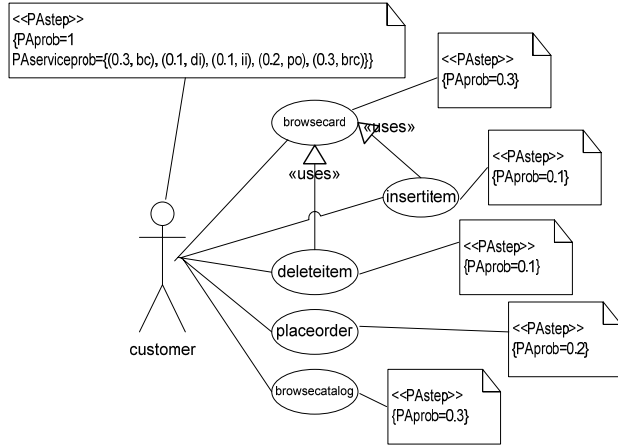
When components are aligned in different columns, they have internal and external arcs to other components of a petri net. Internal arcs enable the movement of component tokens and represent the change of the state within a component. Each incoming place to a transition represents a precondition. This precondition must be fulfilled by a component in order to perform a specific step (fire a transition) of such component. Each outgoing place also represents post

conditions after firing a transition (when a step finishes) within a component. External connecting arcs provide interactions between components and represent the exchange of messages between components. The input arcs to a component can be viewed as the events for a component allowing the activation of a specific step in that component. Finally, the output arcs of a component can be viewed as messages needed to be sent by a client component to server component.

**The implication of refinement**

An important phase of our approach is the refinement process. This process starts with an informal model of the software architecture and adds to it the formal specifications to be ready for being transformed to an executable model. Intuitively, a refinement is the substitution of a part of the specification by another part. Clearly, such substitutions should preserve the semantics of the original specification. In our refinement model, we have defined three different levels of abstraction which are component diagram, formalized component diagram and petri net. The first and third levels have their own correspondence in the underlying model. Since a part of the approach is to refine the first level towards the third level, it is necessary to provide formality at the second level in order to facilitate such a refinement. Furthermore, to provide a formal specification of the component diagram to be compared with its executable model, the provision of the second level is inevitable. In this section, we show the reason that the initial definition of components in all three levels remains consistent and we show how each refinement step helps to move towards a precise specification of components.

Fig. 8. The annotated use case diagram with performance parameters.



At the first level, the components of the component diagram are specified in a general format. If we assume a single thread of control within any component and a flow of control from left to right and top to bottom, a component constrains responsibilities and its required interactions (collaborators) to fulfill its task. However, the precise ordering between such responsibilities remains unclear when an interaction is required. Without considering more formal means for verifying the responsibilities, component diagram would be nothing more than brainstorming. Moreover, a collaborator's name which follows a responsibility shows where and

which collaboration is required. However, it does not provide the exact information about how such collaboration may take place. Clearly, the first level of the component diagram representation needs to be complemented by more precise formal notations. Its formal notation states whether the (implicit) protocols of interaction between the components satisfy the behavioural requirements of the component diagram or not.

The second level is defined as formalized component diagram. This diagram contains the previous definition of components as well as new details about the strict ordering of the responsibilities. Since the responsibilities and their collaborators which were listed in the first step will not be changed in the second one, a component will retain its first characteristics. By providing formal components, we are able to model each responsibility and to achieve a more rigorous definition of our initial representation. Since this level does not alter the collaborators field of components, the new model is still non-specific about how the collaborations will actually be realized. Such clarification needs to be decided by the modeler and be represented as an appropriate formalism for this level.

The third level will be a petri net representation. Semantically, there is no difference between the second and third levels except that a protocol is added to clarify the hidden interaction semantics which exist at the second level. It has been shown by Lauer (1974) that, there is an equivalent petri net representation for every

Fig. 9. A part of the annotated component diagram with performance parameters.

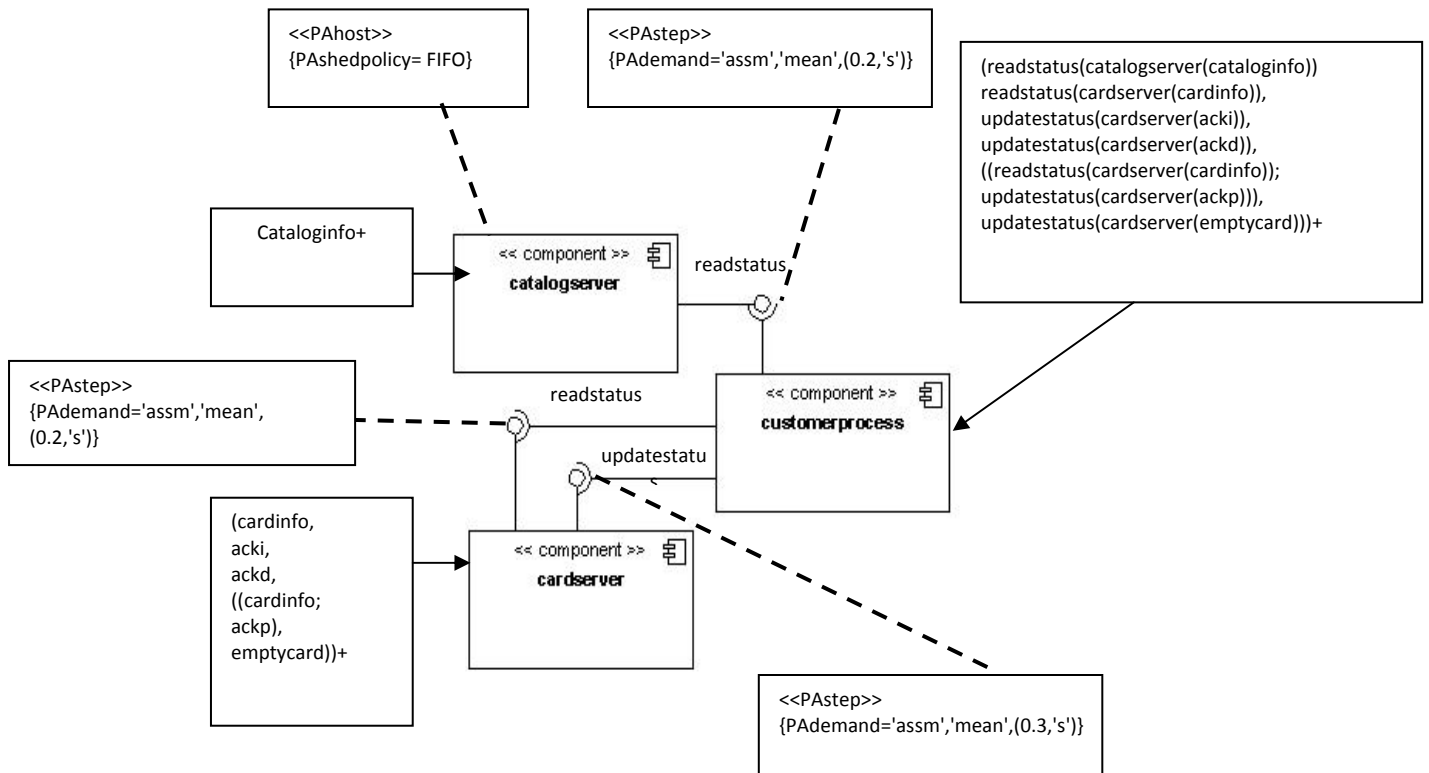
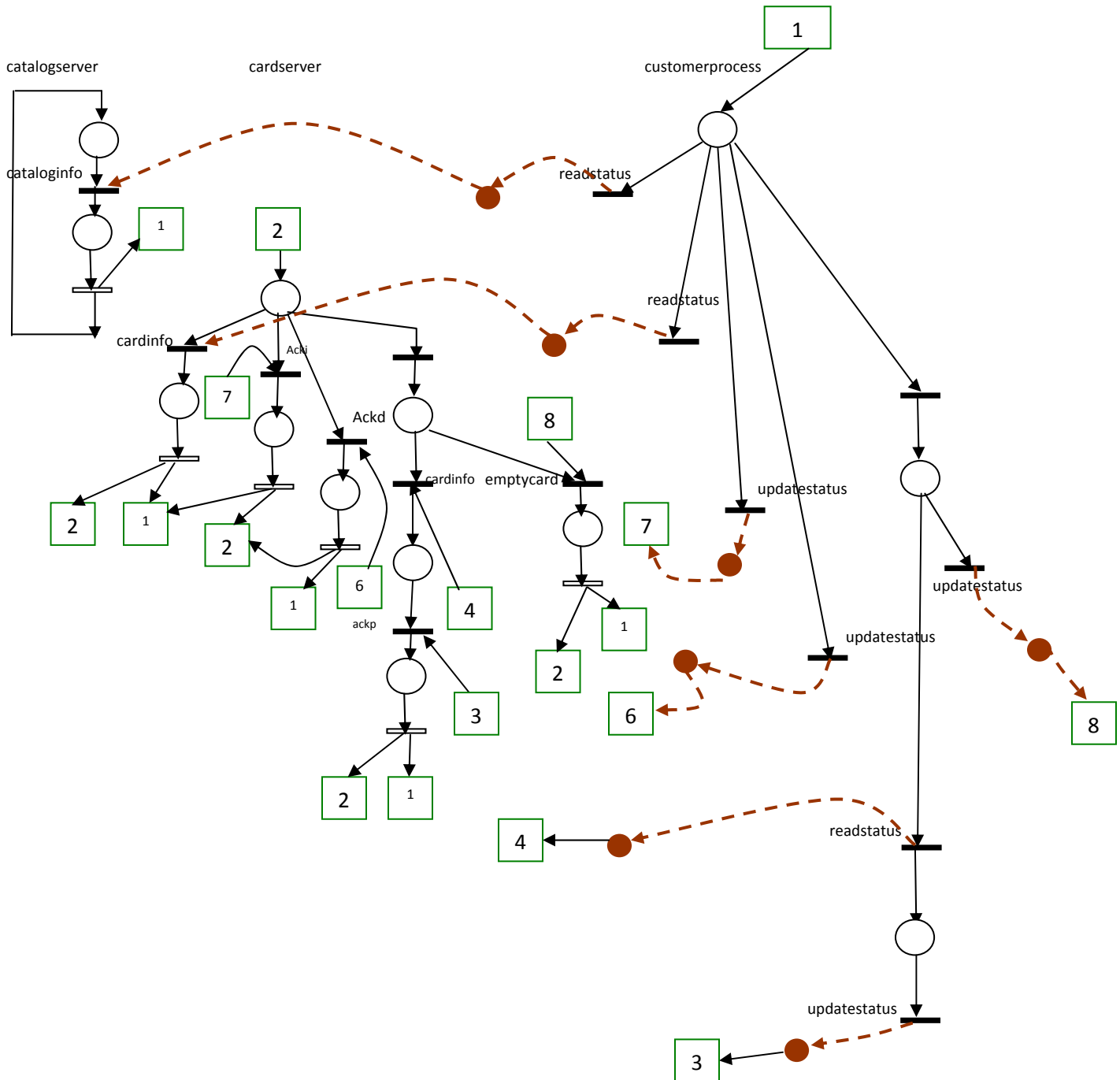


Fig. 10. The equivalent petri net for a part of annotated component diagram.



path expression. Therefore, the third level is an alternative representation of the second level achieved by adding precise interaction semantics.

As you can see, in our approach we provide a clear specification of components not only by including more details through the refinement steps, but also by maintaining the previous information about the original model, which is the notion of semantics preservation (Biberstein & Buchs, 1995).

**Case study**

To represent the usage of our proposed algorithm, in this section we consider an electronic commerce system as an example. In the electronic commerce system, there is a supplier that publishes his catalogue on the web (Di Marco, 2004). The supplier accepts customer orders and delivers the ordered items maintaining all the relevant data. He needs to maintain information on data, on the catalogue and on the orders purchased by his customers.

Each registered customer has a cart where he/she can insert or delete items. The customer can order only if the cart is not empty. The system also allows the customer to monitor the order status and to confirm the delivery in order to permit the payment (Di Marco, 2004). The use case diagram of the e-commerce system is shown in Fig. 8. All the use cases for the e-commerce system have associated only a sequence diagram, and both Insert Item and Delete Item use cases use the Browse Cart use case.

In Fig. 9 we present the UML 2.0 component diagram for the portion of the e-commerce system we consider. This diagram highlights the software components and their required and provided interfaces. The provided interfaces are represented by a circle whereas the required ones are represented by a semicircle (Di Marco, 2004). The relevant operations and their path expressions are specified for each component. As it is shown in Fig. 9, each component of component diagram is refined and path expressions are added to component operations. The Component diagram is annotated, according to the UML SPT profile, in order to introduce performance aspects of the software components and of their interfaces. Our intent is to carry on a performance analysis at the software architecture level when such information is not available, whereas it is guessed by the designers on the basis of their experience.

As described by the annotations on the component interfaces, all the components are queuing centers except the customer process that is a delay center (Di Marco, 2004). The customer process is a component running at the user side. There is an instance of it dedicated to each used logged in the system. This design choice makes the customer process component an unshared resource and hence its execution times constitute delays (Di Marco, 2004). The equivalent petri net according to the proposed algorithm is shown in Fig. 10. Customer process, catalog server and card server which are the components in the petri net model have been represented as separate columns. In this way, we distinguish between the internal arcs which show transition from one state of a component to another state, and the external arcs which show message exchange between two components. Places between such arcs can also be interpreted as the ones which represent the pre-conditions of the firing of a transition.

### Conclusion and further works

In this paper, to fill the gap between software architect and non-functional requirement analyst, a new algorithm is proposed to transform a component diagram to an executable model based on petri nets. Component diagram is one of the software architecture description diagrams. In order to do this transformation, the component diagram first must be refined to specify the execution sequence of operations in components and the relationship between components. Then refined

component diagram can be transformed through our proposed algorithm to an executable model based on petri nets. At the moment, there is no existence of automatic approaches that assist in the whole process of performance analysis. To this respect a relevant effort is still required to software designers attending in the process itself.

Therefore, unlike other approaches that build petri net models as a combination of the hardware platform with software requirements, we consider petri net models as performance models at the level of the software architecture. The definition of a petri net model at the software architecture level cannot be completely specified because of the high level of abstraction. If some parameters or characteristics of the petri net are not specified, it is then up to the designer their definition. The model parameter instantiations correspond to potential implementation scenarios, and the performance evaluation results can provide useful insights on how to carry on the development process. Due to the high level of abstraction, we do not model software resources other than the software components presented in the component diagram. This means that buffers are considered if they are explicitly modeled as software components. In our further works, we will consider the transformation of other software architecture description diagrams to an executable model. Moreover, we can consider the annotation of additional information of non-functional requirements to the software architecture description diagrams as well, so that the resulting executable model can be used for evaluating those non-functional requirements.

### References

1. Andol F, Aquilani F, Balsamo S and Inverardi P (2000) Deriving performance models of software architectures from message sequence charts. Proc. of 2<sup>nd</sup> Int. Workshop on Software & Performance, ACM, Canada. pp:47-57.
2. Balsamo S and Simeoni M (2001) On transforming UML models into performance models. Tech. Report Saladin Project, Proc. of Workshop on Transformations in UML (ETAPS01). pp:1-6.
3. Bass L, Clements P and kazman R (2003) Software architecture in practice. 2<sup>nd</sup> edn., Addison Wesley, Boston.
4. Bastide R and Barboni E (2006) Software components: A formal semantics based on colored petri nets. *Elec. Notes Theor. Computer Sci.* 160, 57-73.
5. Bernardi S, Donatelli S and Merseguer J (2002) From UML sequence diagrams and statecharts to analysable petri net model. Proc. 3<sup>rd</sup> Int. Workshop on Software & Performance, ACM. pp:35-45.
6. Biberstein O and Buchs D (1995) Structured algebraic nets with object orientation. Workshop on Object Oriented Programming & Models of Concurrency at

- 16<sup>th</sup> Int. Conf. on Application & Theory of Petri Nets. pp:131-145.
7. Campbell RH and Habermann AN (1974) The specification of process synchronization by path expressions. *Lecture Notes Computer Sci.* Springer Verlag, 16, 89-102.
8. Clements P, Kazman R and Klein M (2002) Evaluating software architecture methods and case studies. Addison Wesley, Boston.
9. Cortellessa V and Mirandola R (2000) Deriving a queueing network based performance model from UML diagrams. Proc. of the 2<sup>nd</sup> Int. Workshop on Software & Performance, ACM, Canada. pp:58-70.
10. Di Marco (2004) Model-based performance analysis of software architectures, Ph. D thesis.
11. Di Marco A and Inverardi P (2004) Compositional generation of software architecture performance QN models. 4<sup>th</sup> Working IEEE/IFIP Conf. on Software Architecture (WICSA). pp:37-46.
12. Emadi S and Shams F (2008) An approach to non-functional requirements analysis at software architecture level. Proc. of 8<sup>th</sup> Int. Conf. on Computer & Information Technology (IEEE, CIT'2008), Australia. pp:736-741.
13. Emadi S and Shams F (2008) An approach to non-functional requirements analysis at software architecture level. *Engg. J. Islamic Azad Univ., Mashhad Branch* (in Persian), 2(1), 65-79.
14. Emadi S and Shams F (2008) From UML component diagram to an executable model based on petri nets. *Proc. of the 3<sup>rd</sup> Int. Symp. on Transformation Technol.* pp:2780-2787
15. Emadi S and Shams F (2009) A new executable model for software architecture based on petri net. *Indian J. Sci. Technol.* 2(9), 15-25.
16. Emadi S, Shams F and Vaziri S (2008) General syntax for extensions of petri nets. Proc. of 3<sup>rd</sup> Int. Conf. on Mathematical Sciences (ICM2008). pp:1133-1138.
17. Fukuzawa K and Saeki M (2002) Evaluating software architectures by coloured petri nets. Proc. of the 14<sup>th</sup> Int. Conf. on Software Engineering & Knowledge Engineering, Ischia, Italy, ACM. pp: 263-270.
18. Gomaa A, Adam N and Atluri V (2005) Color time petri net for interactive adaptive multimedia objects, Proc. of the 11<sup>th</sup> Int. Multi-Media Modeling Conf. pp:147-157.
19. Kounev S (2006) Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Trans. Software Engg.* 32(7), 486-502.
20. Lauer PE (1974) Path expressions as petri nets or petri nets with fewer tears. *Tech. Report MRM 70*, Computing laboratory, University of Newcastle upon Tyne.
21. OMG (Object management group) (2003) OMG unified modeling language specification, version 1.5.
22. OMG-UML-SPT (2005) UML profile for schedulability, performance and time. Object management group, Version 1.1, Formal/05-01-02.
23. Shams F (1996) Modelling the Behaviour of process using collaborating objects. Ph.D Thesis, University of Manchester, London, England.
24. Silva EA, Almeida H, Silva LD and Perkusich AA (2007) Formal modelling and verification of a software component model using coloured petri nets and model checking. Proc. of 22<sup>nd</sup> Annual ACM Symp. on Appl. Computing, ACM. pp:1427-1431.