A new executable model for software architecture based on Petri Net

Sima Emadi¹ and Fereidoon Shams²
¹Computer Engineering Department, Science and Research Branch, Islamic Azad University, Tehran, Iran
²Computer Engineering Department, Shahid Beheshti University, GC., Tehran, Iran
emadi@srbiau.ac.ir¹; f_shams@sbu.ac.ir²

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 knowledge gap between the architect and the requirements analyst is considered to be significant. In this paper, we propose a new algorithm that enables an architect to transform a software architecture described by use case, sequence and component diagrams into an executable model based on different extensions of Petri Nets. Moreover, we show how to use this Petri net model for simulation, and the implications of this transformation are described completely. Finally, to demonstrate the application of our proposed algorithm, we consider a case study.

Keywords: UML diagrams, Petri Nets, Software Architecture Description.

1. Introduction

One of the most noticeable tasks of an architect is the production of executable models from software architecture description for non-functional requirements evaluation (Clements et al., 2002; Bass et al., 2003). To evaluate non-functional requirements, first an architect should describe software architecture with Architecture Description Language (ADL), Message Sequence Chart (MSC) or Unified Modelling Language (UML). Among which UML is considered to be the most important to describe the software architecture. Then the analyst must evaluate non-functional requirements by an executable model such as Petri nets, queuing networks, process algebras, Markov chain, etc. This kind of evaluation is difficult, time-consuming and expensive.

Different approaches for performance evaluation at software architecture level and comparison of some recently proposed approaches to the transformation of software architecture described by UML diagrams into performance evaluation models are available (Balsamo & Simeoni, 2001) in which different types of UML diagrams were considered and different types of performance models, such as queuing networks, stochastic Petri nets, stochastic process algebras, and simulation models were derived. 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. Andol et al. (2000) have presented an approach to derive automatically a performance evaluation model based on a 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 a queuing network.

Most of these researchers use a queuing model. The quality attributes considered by them are restricted to measuring performance, and are for communication-based architectures, as well. In addition, queuing networks are not suitable for qualitative attributes but very useful in stochastic analysis of any kind of architecture. However, the applicability of this model for different types of non-functional requirements, as well as for 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; Kounov, 2006; Silva et al., 2007). Most of these researchers 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). In our previous works (Emadi & Shams, July 2008; Emadi & Shams, fall 2008), we proposed a framework to fill the gap between the software architecture and analysis of non-functional requirements. 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 (Emadi et al., 2008). According to this framework, the input of our algorithm in this paper is a software architecture described by a set of use case diagrams, sequence diagrams and component diagrams. While 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.

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. The present paper describes the mapping between the component model and colored Petri nets, the chosen behavioral notation. For each component of the assembly, the paper provides a formal behavioral specification. Finally, it describes the denotational semantics that allows constructing automatically a single, unstructured Signal net from the behaviors of all the components and their interconnections. This net describes the behavior of the assembly as a whole. Elkoufli & Rodulf (1998) have transformed a simple use case structure to colored Petri nets. Kamandi et al. (2006) have transformed use case to Object Stochastic Activity Network (OSAN). Different
approaches are used for the transformation of sequence diagrams to Petri nets. In the approach proposed by Bernardi et al. (2002), all structures of the sequence diagram have been transformed to Generalized Stochastic Petri Nets. Ourdani et al. (2006) transformed the simplest structures in the sequence diagram to colored Petri nets. The difference between the two transformations is based on mapping messages as well as conveying them (Bernardi et al., 2002), while in the approach of Ourdani et al. (2006), the transformation is based on message sender and receiver component. The implementation of the second approach seems simple and more understandable.

Furthermore, none of these researchers have utilized use case, sequence and component diagrams for non-functional requirements evaluation of software architecture description based on Petri nets, either. Also, in an executable model based on Petri nets, when we add a quality attribute in order to measure it, 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 is clear, each non-functional requirement problem after the interpretation of the non-functional requirement results, they proceed to develop the software system. Otherwise, to meet the non-functional requirement, they modify the software architecture and re-iterate the non-functional requirement analysis process over the new software architecture. In the software architecture modification, the designers will consider the insights gained from the non-functional requirements analysis.

The rest of the paper is organized as follows: In section 2, an algorithm for the transformation of use case, sequence and component diagrams to Petri nets will be proposed. Section 3 is about the simulation of the Petri net models dynamics. In section 4, the implication of the refinement will be explained. Section 5 presents a case study which illustrates the proposed algorithm. Ultimately, section 6 concludes the paper.

2. The transformation of UML diagrams to Petri nets

As discussed in the previous section, the proposed methodology uses UML diagrams (sequence, component and use case diagrams) to describe the system at the software architecture level. 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 and analysis spaces) to assist the architect in evaluating the process. In this section, we propose algorithms for the transformation of these diagrams to Petri nets.

2.1. The transformation of use case diagrams to Petri nets

In order to transform a use case diagram to a Petri net, the idea presented by Elkoutbi and Rodulf (1998) is applied with some adjustment. In this section, all relations between actor and use case (association) and also the relationships between use cases (include, extend and generalization) have been considered. The use case diagram can be transformed to the Petri net through the following steps:

- The transformation of each use case to a Petri net: In this transformation, each use case is transformed into a Petri Net model. We use one place for each actor and one dark place for each use case. The input for Place is a transition with a guard. The guard includes the calling condition of use case actor1

Fig. 1.a) A use case diagram and the Petri net model for it. b) The extended relation between two use cases and the Petri net model for them
considered. In the next stage, the dark places are replaced with the obtained Petri net from the sequence diagram.

Different states of the message passing can be as follows: The existence of a token in the place of actor means an actor is interacting with the system. The firing of the transition after it means calling for a use case by an actor. The existence of a token in the dark place means use case is ready for its task, and the firing of the transition after it means terminating the task of use case, sending a result and returning to the caller actor. Fig. 1a displays a use case and its transformation to a Petri net. In this figure, time is assigned to transition $t_1$ to display rate and start time of execution of each use case. The associated time to transition $t_2$ displays the time at which the system returns to a state of restarting the actor.

-The transformation of the extended relation between two use cases: In this transformation, when use case 1, $uc_1$, is being executed, the execution of use case 2, $uc_2$, is optional. The condition of this selection is assigned to transitions $t_2$ and $t_3$ in Fig.1b. The existence of a token in the second place means the second use case is ready for its task.

-The transformation of the include relation between two use cases: Based on the semantics of the include relation, the execution of $uc_2$ is mandatory. Several scenarios are assumed for the execution of the use cases $uc_1$ and $uc_2$ (Elkoutbi & Rodulf, 1998; Kamandi et al., 2006). In the first state, $uc_1$ and then $uc_2$ are executed. In the second state, first $uc_2$ and then $uc_1$ are completely done and in the third case some parts of $uc_1$ and then some parts of $uc_2$ are executed. These scenarios are shown in Fig.2a-2b. Fig. 2a shows complete execution of $uc_1$ and then $uc_2$, and Fig. 2b shows the execution of other scenarios.

In Fig.2b some different states of the message passing can be as follows: The existence of a token in the place $uc_1$ means it is ready for its task. It also means that the next use case can be executed or the result is sent and returned to an actor. The condition of selection of these two states is associated with corresponding transition. The existence of a token in the second place means the second use case is ready for its task. It also means that another part of the first use case can be executed or the result is sent and returned to an actor. The condition of selection of these two states is associated with corresponding transition.

-The transformation of the generalized relation between two use cases: The difference between this relation and extended relation is that in the generalized relation, child relation replaces parent relation without interrupting the execution. Fig.3 shows this transformation. In this figure, an extra place is used which assigns selection probability of parent or child use case to the related transition.

A use case also may be called by some actors or vice versa. In this case, probability of these must be assigned to related transitions and their relations.

2.2. The transformation of sequence diagrams to Petri nets

Sequence diagrams are useful design tools because they provide a dynamic view of the system behavior, which can be difficult to extract from static diagrams or specifications.

The aim of this stage is to transform the components and their sending messages to Petri nets. In transforming the sequence diagram to Petri net, the proposed idea by Ourdani et al. (2006) with some modifications was used.

In Sequence diagram, there exist different messages: Synchronous or asynchronous messages and delayed or instantaneous messages. In this paper, all of the existing
messages were transformed into Petri net. Given a UML sequence diagram $sd_i$, the approach taken for its translation into a Petri nets model consists of the following steps:

- Transforming each of the components of $sd_i$, that send or receive each message of the $sd_i$ to a Petri net subsystem. This transformation will map each component to equivalent Petri net subsystem.
- Combining the obtained Petri net subsystems of previous step based on message passing 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 message passing between sender and receiver components.
- Defining the initial marking on the obtained Petri net.

In the following, the details of each step will be explained.

2.2.1. Petri nets representation of components & messages

Fig. 5. Petri net models for sequence diagram structures

$$a_1$$

$$a_2$$

$$b_1$$

$$b_2$$

$$c_1$$

$$c_2$$

$$d_1$$

$$d_2$$

$$e_1$$

$$e_2$$

$$e_3$$

$$ev_1$$

$$ev_2$$

$$opt$$

$$cond$$

$$loop$$

$$while$$

$$repeat$$

$$while$$

$$repeat$$
As mentioned above, transitions denote the activation of operations within components and the places are used to represent pre-conditions to fire a transition. We present the transitions and places of every component of sequence diagram aligned in one column to indicate their belonging to the associated role. Then, an approach to the extraction of the equivalent Petri nets from a formalized component will be elaborated. Let us suppose that message \( m \) can be either synchronous or asynchronous, each of which can be delayed or instantaneous. Fig.4 shows the four possible translations into Petri net of message \( m \).

- **Petri nets representation of asynchronous and instantaneous messages**: In this state, server and client components are displayed in the form of place-transition-place. The message passing between these two components happens through a shared place. This place plays the role of output place for the client component and input for the server component. Fig.4a shows this type of message.

Different states of the message passing can be as follows: The existence of a token on the first place in the client component means the initiation of the sending and in the server component means the waiting for a token on a shared place. The firing of the transition in the client component means the sending, and in the server component means the action receiving. The existence of a token on the second place in the client component means the termination of the sending, and in the server component means the end of action execution. From the system viewpoint, the sending component sets a token in the shared place which means actx action is required. The invocation is valid when the receiving component is in a state where it is allowed to use the existing token in the shared place.

- **Petri nets representation of asynchronous and delayed messages**: In this state, the client component is displayed as in the previous state. But the server component is shown as place-transition-place-transition-place. The second transition is a timed transition with an assigned firing rate. Fig. 4b shows this type of messages.

- **Petri nets representation of synchronous and instantaneous messages**: both the client and the server components are shown in the form of place-transition-place-transition-place and their connection can happens through two shared places. Shared place \( P_1 \), in Fig.4c, transposes action and related data from the client component to the server component, and shared place \( p_2 \) returns the result of the operations. The second place in the figure plays the role of the waiting place for the client component and the provider of the actx action for the server component.

- **Petri nets representation of synchronous and delayed messages**: In this state, the client component is like the previous state and the server component is displayed as place-transition-place-transition-place-transition-place where central transition is of the timed transition. Fig.4d shows this type of message.

- **Petri nets representation of asynchronous and delayed messages**: As mentioned above, each use case can be described with a scenario which is displayed by a dark place in Petri net model. Therefore, it is necessary to substitute resulting Petri net from sequence diagram for any dark place displaying use case. If one or several sequence diagrams are drawn for each use case, the possibility or selection conditions of each sequence diagram should be added to arcs and transitions of Petri nets.

**2.3 The transformation of component diagram to Petri net**

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 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. 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 onto 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 component diagram can be transformed to the Petri net through the following steps:
2.3.1 Refinement of the Components in Component Diagram

It is difficult to characterize the possible behavior of a component through the informal representations. Therefore, to specify the constraints on a component's behavior, we use the path expression formalism (Shams, 1996). 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 (Shams, 1996; Emadi & Shams, Aug 2008). Path expressions are used to specify the exact semantics of ordering between the operations within a component. This ordering prescribes the possible behavior of a component and helps to clarify what should be done in terms of primitive operations such as sequence, alternation, and iteration of operations (Emadi & Shams, 2008).

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 component operations, the name of this operation comes in front of the collaborator components. For example, Fig.6a indicates a refined component diagram. As shown, component c1 includes two operations x1 and x2, which are executed sequentially. To execute x2, y1 from component c2 is needed.

2.3.2 Petri nets presentation 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 presentation, we provide a precise semantics of operations for a component.

We present 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 present 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.6b indicates the equivalent Petri net for each component of Fig.6a. In this figure, separate Petri nets are drawn for components c1 and c2. The component c1 contains two operations x1 and x2. This component first calls x1 and then x2. x2 needs y1 from c2; therefore, the relevant transition must be connected to the component c2.

Fig.7 shows the equivalent Petri nets for the primitive operations, sequence, selection, iteration and concurrency of path expressions (Emadi & Shams, Aug 2008).

In Fig.7a, a sequential path containing two operations has precisely the same behavior as its equivalent Petri net, which contains two transitions connected by a place. Similarly, the other path expression primitives have their own equivalents in terms of Petri nets as shown in Fig.7.

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 presentation 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 onto Petri net.

2.3.3. Petri net representation of the component collaborators

In the 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.

For example, Fig.6c shows a complete Petri net for the component diagram shown in Fig.6a. This figure demonstrates an example of the interaction presentation, which 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.6c, the output of the transition x1 is connected to the input of the transition y1. 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 in a component to the following types:

- Fig.8a shows a typical sequential path expression for the interactions and their equivalent Petri net. 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.8a, when the component c1 and c2 are interacting, the s operation from the component c2 will be used.

- Fig.8b shows a selection path and its equivalent Petri net. Depending on the selection of the actions p or q from component c1, the required transition will be activated. Furthermore, a selection path may not necessarily have interactions on both sides. Fig.8c, shows iteration path and its equivalent Petri net. In this figure, after activating the operation s from component c2, its return point is the place from where the activation of that operation has originated. Fig.8d, demonstrates a concurrent path and its equivalent Petri net. The completion of the two concurrent paths is a necessary pre-condition for activating the action which appears after the concurrent paths. However, in this example, only one interaction was required. In general, two or more concurrent interactions are also possible. As specified before, 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.8d, the parallel operator of the path has been placed on a different line in its component.

3. Simulating the dynamics of the Petri net models

Through this translation, the possibility for the simulation and analysis of the software architecture by architect will be provided. 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 representations of the model.

After providing a Petri net model of the software architecture description, 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.

Simulating the behavior of the components by means of a Petri net tool is a further advantage of providing a Petri net representation for software architecture description diagrams. In this executable model, firing transitions in use case diagram show the start and the end of responsibility of
a use case. Firing a transition in sequence diagram shows the activation of the responsibility of a component and firing a transition in component diagram demonstrates the activation of the responsibility of a component. A multiple thread of execution within a sequence and component diagrams can be simulated, by viewing the movement of tokens through the different components at the same time.

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 software architecture description will assist in its understanding and improving the expected behavior of a component. In fact, initialization provides the starting point for presenting the dynamics of the resultant Petri net. For example, the number of the tokens at the starting points of each component in sequence diagram denotes the number of instances of the components employed to play the corresponding role. The movement of the tokens represents the dynamic behavior of such objects. The shared places between two columns in sequence and component diagrams show the flow of message exchanges. Fig.9a -9c demonstrate a Petri net model of the Fig.1a, 4c and 6c after the initialization of places, respectively.

When in resulting Petri net from sequence and component diagrams, components are aligned in different columns, they have internal and external arcs to other components of a Petri net. Internal arcs which enable the movement of component tokens represent the change of the state within a component.

Each incoming place to a transition presents a precondition. This precondition must be fulfilled by a component in order to perform (fire a transition) a specific step 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.

4. 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 the formal specification to it 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. Obviously, such substitutions should preserve the semantics of the original specification.

In our refinement model, we have defined two different levels of abstraction for use case and sequence diagrams which are software architecture description and Petri net model. These levels have their own correspondence in the underlying model. These two diagrams and Petri nets represent the two different levels of abstraction which exist in our refinement model. Since the intention of the approach is to refine the first level towards the second level, it is necessary to isolate different messages between components and relations between use cases in sequence and use case diagrams at the second level, respectively.

Also in our refinement model, we have defined three different levels of abstraction for component diagrams which are component diagram, formalized component diagram and Petri nets. 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 in order to compare it with its executable model, the provision of the second level is inevitable. In this section, we provide the reason that the initial definition of components at all three levels remains consistent and demonstrate how each refinement step helps to move towards a precise specification of components. 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 behavioral 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 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 when a protocol is added to clarify the hidden

Fig. 10. a) The use case diagram b) The sequence diagram for ATM c) The component diagram
interaction semantics which exist at the second level. It has been shown in (Shams, 1996) that, there is an equivalent Petri net representation for every path expression. Therefore, the third level is an alternative representation of the second level achieved by adding precise interaction semantics.

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.

5. Case study

To represent the usage of our proposed algorithm, in this section we consider a single Automated Teller Machine (ATM) as an example. The use case diagram of an ATM system is shown in Fig. 10a. The relation between the use cases Balance and Identify is of includes type. On the other hand, the relation between "Print" and "Balance" is of extend type. However, in order to check the balance, the identification of the customer is mandatory. It shows the difference between these two types of relations. One of the sequence diagrams of ATM system corresponding to the use case "Identify" is shown in Figure 10.b. This diagram is used when the PIN entered by a customer is valid; the identification will be successfully done. A part of ATM component diagram is shown in Figure 10.c, as well. The relevant operations and their path expression are specified for each component. As it is shown in this figure, each component of component diagram is refined and path expressions are added to component operations. For example, in this refinement, the sequence of executing the "identify" use case is added to the operations in each component as path expression.

According to the proposed algorithm, the equivalent Petri net of these diagrams are shown in Fig.11. ATM, bank, account and customer which are the components in the Petri net model have been presented as separate columns in sequence and component diagrams. Each column presents a separate component. In this way, we distinguish the internal arcs which show transformation 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 present the pre-conditions of the firing of a transition.

As stated before, each operation of a component in component diagram is modeled as a transition in the Petri net representation. After considering the delegation of components to their agent components, in this step each interaction is realized by using its corresponding message passing representation in terms of Petri net. Also each component of the sequence diagram is modeled as place-transition-place in the equivalent Petri net representation and the interactions between them are realized by using message passing. In use case diagram, customer place selects one of these use cases: balance, withdrawal or deposit. In each use case, the identification of the customer is mandatory.

6. Conclusion

To fill the gap between software architect and non-functional requirements analyst, a new algorithm is proposed to transform a software architecture description to an executable model based on Petri nets.

We employed use case, sequence and component diagrams for software architecture description. In order to execute this transformation, these diagrams first must be refined to specify the execution sequence of operations in components and the relationship between components. Then refined diagrams can be transformed through our
At the moment, automatic approaches which assist in the whole process of non-functional requirement analysis do not exist. A relevant effort is still required to software designers attending in the process itself. Therefore, unlike other approaches which build Petri net models as a combination of the hardware platform with software requirements, we consider Petri net 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 non-functional requirement in the Petri net are not specified, their definition is then up to the designer. The model parameter instantiations correspond to potential implementation scenarios, and the non-functional requirement 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 diagrams. This means that buffers are considered if they are explicitly modeled as software components.

In our further researches, 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, the resulting executable model can be used for evaluating those non-functional requirements.

References