An Approach for Modeling and Analyzing Dynamic Software Architectures

Junhua Ding
Department of Computer Science
East Carolina University
Greenville, NC, USA

Abstract— **Software architectures define the overall structure of software systems as composition of interacting components connecting through connectors. As the foundation to the development of software systems, the correctness of the software architecture is critical to the quality of the final product. Formally modeling and analyzing software architectures is an effective way to ensure the correctness of the software architecture. Many formal specification and analysis approaches have been proposed during past three decades. However, the focus of the majority of the approaches is on the static software architecture, which doesn't change the composition of components during a computation. As cloud computing has been widely adopted as a new computing paradigm, the dynamic software architecture that changes the composition of components during a computation becomes an important research topic. Although research work on specification and analysis of dynamic software architectures was published 20 years ago, building current distributed systems requires better scalability and usability for the modeling and analysis approach. In this paper, the software architecture is modelled using a two-layer higher Petri nets extended with communication channels called CPrT. CPrT nets model static and dynamic software architectures using a uniform formal notation. Its graph notation is easy to use and its executable is necessary for developers to build complex models. Its communication channels that are used for modeling the dynamic composition of software architectures implement the channels in pi-calculus. The semantics of CPrT nets can be described through transforming them into regular Petri nets. The analysis of CPrT nets is conducted using model checking with its tool SPIN.**

Keywords- software architeture; dynamic configuration; Petri net; communication channel; model checking

I. Introduction

Software architecture is an overall structure of a software system, which consists of a group of interacting components and the connections among the components in addition to the constraints applying to the connections [1]. It is the foundation of software product lines and the development of software systems. Therefore, the correctness of software architecture is important to the quality of software systems. Formal modeling and analysis of software architecture offers a rigorous way to ensure the correctness of software architectures. Many articles on formal specification and analysis of software architecture have been published during past 25 years [3][1][7][9][22][4][11][12][19]. However, majority of the publication is about the specification and analysis of the static software architecture, which doesn't reconfigure its composition of interacting components during the course of a single computation [2]. As cloud computing has been widely adopted as a new computing paradigm, the dynamic software architecture that changes the composition of components during the course of a single computation becomes an important research topic [2]. For example, a cloud computing service may include several alternative backup subsystems to be selected at real time, and it may include customized information retrieving services to be selected based on real time contexts. A mobile agent system that hosts the running of incoming mobile agents is a typical system that has a dynamic software architecture [7]. Although research work on specification and analysis of dynamic software architectures was published 20 years ago [2], building modern distributed systems like cloud computing systems and mobile computing systems requires better scalability and usability for the modeling and analysis approach. Modeling and analysis of dynamic software architecture is still an active research topic, and current research focus is on the application of research results to cloud computing systems and mobile computing systems [3][20][19]. However, current approaches for formally modeling and analysis of dynamic architectures are difficult to use due to their non-executable. Running the architectural models and automatically verifying the execution results is extremely effective and practical for ensuring the quality of the architecture. In addition, a graph notation is relatively easier to use and its architectural models are also easier understood. A tool to support the automated analysis is also necessary to the analysis of software architectures. Model checking is a powerful analysis technique that can be adopted for analyzing software architecture. In this paper, we model software architectures using a two-layered higher Petri nets called CPrT nets that are extended with communication channels to a high level Petri nets [7]. CPrT nets model static and dynamic software architectures using a uniform formal notation. Its graph notation is easy to use and its executable is necessary for developers to build complex models. Its communication channels that are used for modeling the dynamic configuration of software architectures implement the channels in pi-calculus [17]. The semantics of CPrT nets can be described through transferring them into regular Petri nets. The analysis of CPrT nets is conducted using model checking with its tool SPIN [13].

In this research, we use mobile computing systems as an example to illustration the approach. Mobile computing systems are distributed systems with moving code that has the ability to move actively from one computer to others in a network. A mobile agent may move in or out from its host system during the course of a single computation so that the software architecture is reconfigured at real time. The dynamic configuration of software structures brings grand challenges for building high quality systems. It would be necessary to model and analyze software architectures to detect and eliminate design errors as early as possible so that to avoid costly fixes at later development stages, and reduce overall development cost and improve the system quality. Petri nets [18] as a well-studied formal method with graphical and mathematical notations are noted for its many advantages in specifying and analyzing concurrent systems, and they are also a promising tool for studying systems with dynamic software architectures that are characterized as being concurrent, asynchronous, distributed, and non-deterministic [22]. Predicate/Transition (PrT) nets are high-level Petri nets that are especially suitable for modeling computing systems with dynamic software architectures due to their similarity to the logic mobile computing system, and the efficient reachability analysis [8][22]. However, the communication among the communication parties in mobile computing systems is built dynamically at runtime, and the communication structure could be reconfigured by the moving code during the runtime. The communication channels in CPrT nets can easily model the mobile communication among mobile agents and their host systems. Model checking as an automatic analysis technique for verifying finite state concurrent systems has been successfully used for verifying the design of mission critical systems, complex sequential circuits, communication protocols and many other systems [5] for over 30 years. In this paper, we use model checking specifically model checker tool SPIN to analyze CPrT net models.

The rest of this paper is organized as follows: Section 2 presents the formalism for modeling dynamic software architectures. Section 3 describes the modeling and analyzing dynamic configuration of software architectures. Section 4 reviews the related work. Finally, we outline our conclusion as well as future work in Section 5.

II. A Predicated/Transition Net with Channels

A. An Example of PrT Nets

Predicate/Transition (PrT) nets are a type of high level Petri nets and they are good for specifying concurrent systems. The definition of PrT nets can be found in [21]. Fig. 1 shows a simplified PrT net model for 5 dining philosophers' problem (*i.e.*, without considering deadlock and starvation issues). The model includes transitions *Pickup*, and *Putdown* represent the action for picking up chopsticks and putting down chopsticks, respectively. The distribution of tokens in places *Phi*, *Chop* and *Down* represents the three states of each philosopher: *thinking*, *full* and *eating*, respectively. Places *Phi* and *Chop* define philosophers and chopsticks, and the tokens are defined by nature numbers. Place *Down* define the state that a philosopher has put down his or her chopsticks, therefore, its token includes a philosopher and his/her two chopsticks. Transition *Pickup* includes two input places, which are *Phi* and *Chop*, and one output place, which is *Down*. The guard condition of transition *Pickup* is defined on the relation of the tokens in place *Phi* and *Chop*: which is $x{=}c\&\&d{=}(x{+}1)\%5$, which says that a philosopher must have both of his or her left and right chopsticks before he or she can eat (i.e.: *pickup*) The guard condition in transition *Putdown* is defined on the relation of the tokens in place *Phi* and *Chop*: which is $x{=}c$, which says a philosopher who has to puts down both chopsticks together.

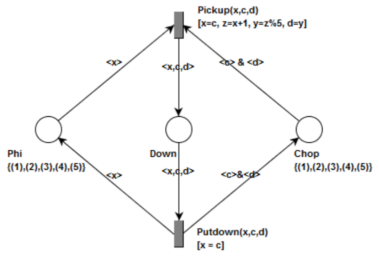

Figure 1. A PrT nets model of the problem of dining philosopher

B. PrT Nets Extended with Channels

In this research, a dynamic channel is introduced into PrT nets to model dynamic interactions and communications between nets. A channel is a special relation that is defined in transitions for sending and receiving messages between nets. The channel concept is borrowed from pi-Calculus [17], and the definition of the channel and the definition of the PrT net extended with Channels called CPrT nets can be found in [7].

A CPrT net model may include several nets and they communicate through channels at runtime. An output channel identifier could be a variable that is instantiated with a concrete value and matched to an input channel at runtime. The communication topology is dynamically built according to the context of the communication transitions.

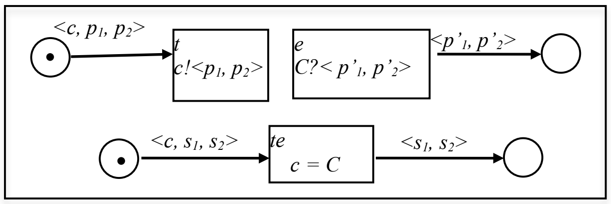

Figure 2. The runtime communication between dynamic channels

A CPrT net can be transformed into an equivalent PrT net. Therefore, the semantics of CPrT nets can be defined using PrT nets. The basic rules of PrT nets are then applied to CPrT nets as well. A CPrT net can be transformed into an equivalent PrT net through combining matched input and output channel transitions. When a transition with an output channel is merged with a transition that has a matched input channel, the input and output flows of the matched transition are the union of the corresponding flows of the two matched transitions. The guard condition of the new merged transition is defined by the conjunction of the guard expressions of the matched transitions as well as the expression defining the communication between the input channel and an output channel. It is important to ensure the set of variables in the matched transitions are different before the transitions can be combined together so that the same variable name at different transitions won't cause any conflict [6]. The transformation idea is illustrated in Fig. 2.

The firing sequence of the two communication transitions with matched input and out channels is described as follows:

1. Assume a CPrT net model includes two nets: N_1 and N_2. Transition t is enabled under marking M_1 in net N_1, and

transition e is enabled under marking M_2 in net N_2. Transition t and e both are enabled under marking $M = (M_1, M_2)$. Transition t has channel $c!<p_1, p_2>$, and the transition e has channel $C?<p'_1, p'_2>$.

2. Under marking (M_1, M_2), the value of output channel variable c equals to channel C. The numbers of input parameters of input channel $C?<p'_1, p'_2>$ and output channel $c!<p_1, p_2>$ are equal, and type of each corresponding parameter in both channels is compatible, *i.e.* $dom(p_1) \subseteq dom(p'_1)$, $dom(p_2) \subseteq dom(p'_2)$.

3. Transition t and e fire together as an atomic transaction. Token $<p_1, p_2>$ is moved from the input places of t to the output places of e following regular PrT firing rules.

4. The enabling and firing sequences as well as the firing result of the CPrT net are same as its corresponding transformed PrT net. When t and e fire, a new marking M' is produced. Formally, $M'(p) = M(p) - \{l/\theta: l \in L(p, t)\} - \{l/\theta': l \in L(p, e)\}$ for any $p \in {}^\bullet t \cup {}^\bullet e$, and $M'(p) = M(p) \cup \{l/\theta: l \in L(t, p)\} \cup \{l/\theta': l \in L(e, p)\}$ for any $p \in t^\bullet \cup e^\bullet$.

5. If two or more output channels match to an input channel under certain marking in a CPrT net, then only one output channel is selected to match the input channel. Which output channel to be selected is non-deterministic [5].

In order to specify a mobile computing system, the concept of "net within net" proposed in EOS [15] is introduced to PrT nets for building two-layer PrT nets. In "net within net", a token could be defined as a net, and a net may include a token that is a net. A PrT net that is wrapped as a token is called a token net, and a PrT net that include any token net is called a system net.

Definition 1 (Two-layer CPrT Net). A two-layer CPrT net is a tuple $STN = (SN, TN, \rho)$, where:

- SN is a finite set of system nets, $SN = \{SN_1, SN_2, ..., SN_n\}$, and SN_i $(1 \le i \le n)$ is a CPrT net, $SN_i = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$.
- TN is a finite set of token nets, $TN = \{TN_1, TN_2, ..., TN_m\}$, and TN_i $(1 \le i \le m)$ is a CPrT net, $TN_i = (P', T', F', \Sigma', L', \varphi', M'_0, C', W')$.

$$TN_i \in \bigcup_{i=1}^{n} (SN_i \cdot \Sigma) \qquad (1)$$

- $\rho \subseteq W \times W'$ is the occurrence relation between channels.

The net occurrence in different layers interact each other through channels. We define the marking of the system net as M, and the marking of a token net as M', so that the marking of the CPrT net is (M, M'). The interaction occurrence between a system net and a token net is completed through the communication of matched input and output channels in a system net and its token net under marking (M, M'). The marking of the two-layer net is updated when the transitions fire: $(M, M')[(t, t') > (M_1, M_1')$, where t is the fired transition in the system net and its marking is updated from M to M_1: $M[t > M_1$; and t' is the matched transition in the token net and its marking is updated from M' to M'_1: $M'[t' > M'_1$.

III. Modeling and Analyzing Dynamic Configuration

In this section, we illustrate the approach for modeling and analyzing dynamic software architectures through case studying a mobile agent system. A mobile agent system includes a host system which can host the running of incoming mobile agents. A mobile agent is a program that has its computation ability and itinerary for moving among networked hosts. When an agent arrives at a host and is authorized for running, it can run within the host environment [7][22]. The moving in or out of a mobile agent from a host requires the re-composition of the interacting components in the software architecture of mobile agent systems. The software architecture of a mobile agent system includes two levels: the system level and the interaction level. In the system level, each agent is considered as a token within the system net which models the host, and the location of the system net is predefined. If we consider the dynamic configuration of the host systems such as some host systems may join in or leave during run time, we change host nets with an additional Boolean variable on the inscriptions of channel transitions to indicate whether the system is active or not. The variable is part of the guard condition and it disables the channel transition when it is *false* so that the system net won't receive or send messages from/to other systems or agents. From the system point of view, the system is disabled. In this paper, we only consider one host system since it is not difficult to be extended to multiple systems. At the interaction level, the software architecture is dynamically configured at run time when the system net connects with different agent nets. Agent nets communicate with other objects through channels, and each agent has one unique input interface to receive messages from others so that it guarantees messages to reach the correct destinations. We call the channel as the agent channel, and its value is a dummy constant when it is defined in the template of agent nets. The dummy value is instantiated by a unique value same as the instance identifier when an instance is instantiated from a template. In order to define the dynamic reconfiguration of the software architecture, we introduce a concept called configuror to remember current active agents in each host net. Based on the system configuror, one can reconstruct and analyze the snapshot of the software architecture.

A. System Configuror

There are only finite numbers of object nets (*i.e.* the instantiated agents) in a system net at any time, so we can transform the dynamic view of a software architecture into a static view to study interaction properties. The key issue is how we can transform a dynamic view into a static view at run time. A configuror defines the configuration of object nets with their system nets. We do not add any configuror to CPrT nets, but it is used for describing the system configuration when we analyze the models. The configuror is responsible for defining the dynamic reconfiguration of the software architecture. Each system net has a configuror, which consists of agent instance identifiers, agent types (agent nets) and agent itineraries. When an agent is created, it is assigned with an itinerary that decides the visiting path of the agent. Based on the knowledge or itineraries of an agent, it may dynamically update its itineraries at run time. The configuror of the software architecture is the combination of configurors of all host nets.

Definition 2 (Configuror) The configuror of each system net is a list *CON = {c_1, c_2, ... c_n}*, where:

- $c_i = (AN_i{\cdot}ID, AN_i{\cdot}TYPE, AN_i{\cdot}KB)$, $1 \leq i \leq n$. The *n* is the number of agent instances in the system net. AN_i is the agent instance in the host net, and $AN_i{\cdot}ID$ is the instance identifier of AN_i, $AN_i{\cdot}TYPE$ is the instance type (the name of the template net of AN_i), and $AN_i{\cdot}KB$ is the instance itinerary of AN_i.

When a host net receives an agent, the agent location is updated to the location of the host system. Then it is put into special place p_a in the host net, which is the only place the agent can update its states as soon as the host system starts it. When an agent moves it out from the host, it updates its location according to its itinerary and terminates its execution until the destination host accepts it. An agent system can generate agents or instances of agent nets (we call instances of agent nets as object nets) according to existing agent types (templates of agent nets), but each object net has its unique identifier and itinerary. When a host net receives or generates an object net, the configuror adds the object into its list, and it removes the object from its list when an object leaves the host net. The configuror is easily constructed from agents within p_a. The static view of interaction between host nets and object nets is a net composing the host net and a group of object nets within the host net. Fig. 3 shows the basic idea to analyze the dynamic configuration of host nets.

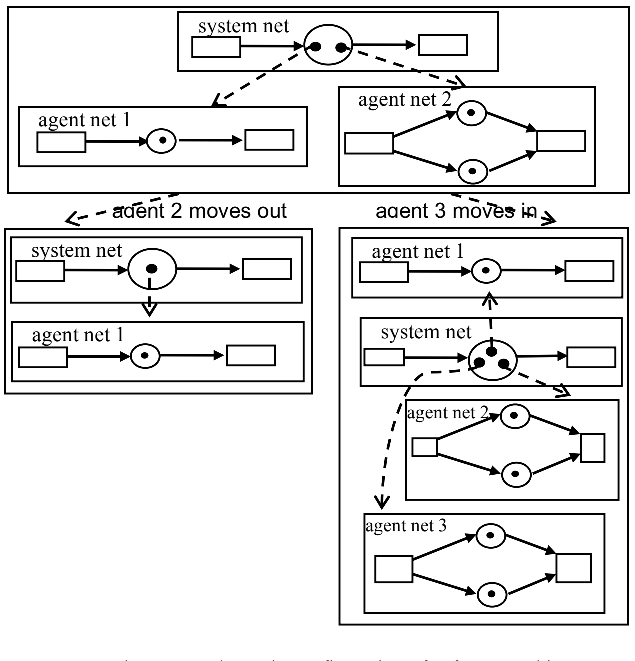

Figure 3. A dynamic configuration of software architecture

In Fig. 3, the system net or host net has two agents within its place p_a, so that its configuror includes the information of these two agents, which can be used to construct the static view of the host model which includes one host net and two agent nets. When one agent moves it out, the configuror removes the agent (agent net 1) from its list, so that the static view of the current host model includes the host net and an agent net. When the host net receives an agent (agent net 3), the configuror adds that agent information into its list, and then the static view of the current host model is the composition of the host net and three agent nets. Based on static views and configurors, we can analyze the dynamic reconfiguration of the software architecture of mobile agent systems.

B. Analyzing Dynamic Configuration

Analyzing the interaction between a system net and its agent nets is implemented through transforming the dynamic model into the static model according to the configuror. All object tokens (the instances of agent nets) in the system net are unfolded as agent nets with states, and these nets consist of a logical whole net even if they may not be connected with arcs, they are logically connected with channels. The analysis is conducted based on the nets and configuors. The occurrence rules of the interaction view are the same as the semantics and analysis of the regular two-layer CPrT nets. The marking of the whole net is the combination of the marking of each net. When an agent moves it out from the host net, the configuror removes that object from its list and the corresponding object net is removed from the interaction view or the whole net. When an agent moves it into the system, the configuror adds that object to its list and the corresponding object net is added into the interaction view or the whole net.

Definition 3 (Interaction view): An interaction view of a software architecture of a mobile agent system is a tuple *IV = (SN, AN, CON),* where*:*

- *SN* is a system net, $SN = (P, T, F, \Sigma, L, \varphi, M_0, C, W)$.
- *AN* is a finite set of object nets, $AN = \{AN_1, AN_2, ..., AN_n\}$, $AN_i = (P_i, T_i, F_i, \Sigma_i, L_i, \varphi_i, M_{i0}, C_i, W_i)$, $1 \leq i \leq n$, $AN \subseteq \Sigma$.
- *CON is the configuror of SN.*

Only one system net is considered in the interaction view of the software architecture of mobile agent systems, where the moving in and out of mobile agents causes the re-composition of the host systems and different agents. The re-composition of components is defined by the dynamic configuration.

Definition 4 (Dynamic configuration): The dynamic configuration of the software architecture of a mobile agent system is defined on the dynamic changes of configuror of the host net. A dynamic configuration is *IV = (SN, AN, CON),* where*:*

- When an agent AN_k moves in to *SN*, $AN_k = (P_k, T_k, F_k, \Sigma_k, L_k, \varphi_k, M_{k0}, C_k, W_k)$, then $AN_k \in P$, $CON = CON \cup \{c_k\}$, and $c_k = (AN_k{\cdot}ID, AN_k{\cdot}TYPE, AN_k{\cdot}KB)$.
- When an agent AN_k moves out from *SN*, $AN_k = (P_k, T_k, F_k, \Sigma_k, L_k, \varphi_k, M_{k0}, C_k, W_k)$, then $AN_k \notin P$, $CON = CON \setminus \{c_k\}$, and $c_k = (AN_k{\cdot}ID, AN_k{\cdot}TYPE, AN_k{\cdot}KB)$.

The occurrence rules and communication between object nets and the system net follow the definitions in CPrT nets.

C. *Checking Dynamic Configuration using SPIN*

In this section, we discuss the approach for model checking CPrT net models of mobile agent systems using model checking tool SPIN [13]. Since it is infeasible to check a model that has infinite states using model checker SPIN, it is necessary to convert the model that has infinite states into a model that has only finite states but the change doesn't affect the properties to be checked. A CPrT net model has to be converted into an equivalent Promela program for SPIN checking, and properties to be checked are defined as the correctness and other claims in the Promela program. Some important system properties are defined as *never* claims translated from LTL formulas. Individual system nets and token nets are first checked independently, and system properties are checked on the system net. A general procedure for model checking the dynamic software architecture using SPIN is defined as follows.

1. *Transform models.* Any CPrT net to be checked is transformed into a PrT net.

2. *Reduce states (convert a model with infinite states into a model with only finite states).* First, each place p in a model is converted into k-bounded, and then the type of each place variable is defined as an enumerable data type with finite number of elements. k is predefined based on system requirements.

3. *Specify properties to be checked.* After the system behavior model B is specified in CPrT nets, each interested system property S is defined in LTL. The model checking procedure is to check property specification S over behavior models B.

4. *Translate a CPrT net model into a Promela program*.

The procedure for transforming a CPrT net into an equivalent Promela program for model checking using SPIN is described as follows.

1. *Structure of the Promela program.* A whole CPrT net model of a system is transformed into a Promela program, and each individual CPrT net in the CPrT net system model is converted into a Promela process. Each Promela program includes sections for data type definitions, global variable declarations, definitions of processes, process *init* for defining the program initialization, and a *never* claim for specifying a property to be checked. Place types are defined in the section of data type definition, and global variables are defined in the global variable declaration. The firing rules of each net is defined by the transition relations in the Promela process.

2. *Specification of the state variables.* Each place in a net is defined as a variable in the data type definition section in the program. The value range of each variable is defined by the value range of the marking of the place. The value range of a variable is the number of possible markings of the corresponding place. If place p is k-bounded and $|\varphi(p)|$ defines the number of possible values of a token in p, then the number of possible markings of place p is $\sum_{i=0}^{|\varphi(p)|} k^i$. The place variable of place p in the Promela program has the form:

$$p : 0..\sum_{i=0}^{|\varphi(p)|} k^i - 1$$

In this way, a predicate symbol is defined as a set of proposition symbols, which can be applied for each place p that is bounded and $|\varphi(p)|$ is finite [7].

3. *Definition of the initial states.* Initialize each variable in the program with a value that is the initial marking of the corresponding place in the net, and the initialization is assigned in *init* process. *init* process invokes each net process with initial values of the input parameters that are all places in the net, and these processes are running in parallel. The fairness of the running of the processes is ensured by model checker SPIN.

4. *Specifications of the transition relations.* There are two types of transitions in CPrT nets, one is transitions that have channels, and the other is regular PrT net transitions. The two types of transitions are processed differently.

4.1. Transferring a transition that doesn't have a channel. Each transition in a net is converted into an atomic statement within a process in the Promela program, and the atomic statement specifies the firing rules (*i.e.* the inputs, outputs and constraints of a firing) of the transition. The atomic statement includes a group of *case* statements. The condition of each *case* statement specifies one possible input of the transition, and the body of the statement defines the relation between input and output. The number of the *case* statements of each transition is the permutation of input variable values in the inscription expressions of the input arcs of the transition. Each case statement is fairly simple based on the net model so that the transformation can be partially automated. The communication between processes are implemented through global variables, and the synchronization between communication transitions are guaranteed with additional global Boolean variables.

4.2. Transferring a transition that has a channel, We first only consider the channel expressions in the transition inscription expressions of the transition that has a channel. Then each channel is then declared as a global variable, whose data type is defined by all possible values of the variable (finite number of values). In CPrT nets, channel variables share names with their input inscription. However, each channel is declared with a unique variable. The variable number is the number of possible values of the channel variable in the net. All of these variables have the same type, which is same to the type of input or output parameters of the channels. Next, we define the transition relations. For output channels, when the transition fires, some channel variable is assigned with values according to the outputs of the transition. Such as one channel has three possible values, P_1, P_2 and P_3, if the output value, which is assigned to the channel in the net, is P_2, then value of P_2 is updated with the value of the output parameter, but P_1 and P_3 do not change. For input channels, according to input tokens and inscriptions on input arcs, we chose one channel variable as part of input conditions of the transition. For example, the channel has three possible values, P_1, P_2 and P_3, and if input tokens instantiating the current input channel is P_2, then P_2 is chosen as part of the input condition of the transition. When the input transition fires, value of P_2 is updated.

5. *Specification of properties to be checked. never* claim in Promela program is used for specifying system properties to be checked. Some properties such as reachability can be defined with *accept-state* labels in the Promela program. Other Promela constructs such as *basic assertion*, *end-sate labels*, *progress-state labels*, and *trace assertions* can be also used to specify interested properties to be checked.

IV. Related Work

Software architectures are an essential part in every phase of software development [9]. Many researchers have proposed approaches and built tools for modeling and analyzing software architectures in order to improve the rigorousness of the analysis and confidence of the quality of the architectural model. Garlan [9] has summarized the representative results of formal modeling and analysis of software architecture. Allen and Garlan [1] described a formal basis for an architectural connection, which has become the one of the most important work on formal modeling of software architecture. Allen *et al.* [2] introduced an approach for formally modeling and analyzing dynamic software architectures, where the architecture is modeled using Architecture Description Language (ADL) Wright, and the analysis is completed through formal verification based on CSP semantics. Comparing to Wright approach, our approach is easier to use due to the graph notation and executable of CPrT nets and model checking of CPrT nets is fully automated. Recently, Sanchez *et al*. [19] proposed an approach for modeling and analyzing dynamic software architectures based on ADL ARCHEY, where the reconfiguration of architectures is defined with constraints. The formal analysis is conducted directly on the constraints. Model checking has been reported for formally analyzing software architecture. He *et al*. [12] reported techniques for formally analyzing Petri nets using model checking and formal proof. Ding and He proposed an approach for modeling checking a type of high level Petri nets [7]. Several other researchers defined an executable semantics for software architectural modeling and analysis through simulation and/or formal verification [16]. Garlan *et al*. [10] introduced a reusable generic framework for modeling and checking the model using model checker SMV. Kim and Garlan [14] also investigated how to analyze software architecture using Alloy.

Mobile computing systems are representative systems that have dynamic software architectures. Petri nets have been used for modeling and analysis of mobile computing systems, such as the Logic Agent Mobility (LAM) [22] was modeled using a type of two-layer PrT nets. In LAM, connectors were introduced in the two-layer PrT nets to support the communication between nets. However, the composition of several PrT nets in different layers using connectors is difficult and the whole model that consists of several nets and the connector could be very large. “nets within nets” style was first introduced in EOS [15], and it was introduced to CPrT nets for modeling the communication between nets in different layers. Channels for Petri nets were first introduced to colored Petri nets in reference [6], but the channels in CPrT nets are more dynamic with high flexibility for specifying dynamic properties in software architectures. The pi-calculus is the first language that offers features for specifying process movement across channels, and the semantics of channels in CPrT nets is same to the channels in pi-calculus [17].

V. Summary and Future work

In this paper, we first introduced an approach for modeling dynamic software architectures using a two-layer high level Petri nets called CPrT nets, which are extended with communication channels. A dynamic software architecture is modeled as a system net that represents the system running environment and a group of agent nets that model the components that are dynamically connected to the system net during the course of a single computation. The dynamic configuration of interacting components is defined in configurors. The analysis of dynamic software architectures is conducted by model checking with tool SPIN through transforming a CPrT nets model into a Promela program. Both the static and dynamic software architectures can be modeled using CPrT nets. The graph notation of CPrT nets is easy to use and its executable is necessary for developers to build complex models. The communication channels of CPrT nets that are used for modeling the dynamic configuration of software architectures implement the channels in pi-calculus. In the future, we would like to investigate the application of the approach for modeling and analyzing enterprise cloud computing systems.

Acknowledgments

This research is supported in part by grant CNS-1262933 and CNS-1560037 from the National Science Foundation.

References

[1] R. Allen, D. Garlan. “A formal basis for architectural connection.” *ACM TOSEM* 6 (3), pp. 213–249, 1997.

[2] R. J. Allen, R. Douence, and D. Garlan, “Specifying and Analyzing Dynamic Software Architectures”, *Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98)*, Lisbon, Portugal, March 1998.

[3] B. Abolhasanzadeh , S. Jalili, “Towards modeling and runtime verification of self-organizing systems”, *Expert Systems with Applications: An International Journal*, v.44 n.C, p.230-244, 2016.

[4] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. “A survey of self-management in dynamic software architecture specifications”. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems* (WOSS '04), pp. 28-33, 2004.

[5] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking,* The MIT Press, Cambridge, 1999.

[6] S. Chrisensen, N.D. Hansen, “Coloured Petri Nets Extended with Channels for Synchronous Communication”. In *Proceeding of Application and Theory of Petri Nets* (1994), pp. 159-178.

[7] J. Ding, X. He. “Formal Specification and Analysis of an Agent-Based Medical Image Processing System.” *Intl. Journal of SEKE*, Vol. 20, No. 3, pp. 1 – 35, 2010.

[8] H. J. Genrich, “ Predicate/Transition Nets”. *Petri Nets: Central Models and Their Properties*, W. Brauer, W. Resig, and G. Rozenberg, eds., (1987) pp. 207-247.

[9] D. Garlan, “Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events”, in *Formal Methods for Software Architectures, LNCS,* Vol. 2804, pp. 1 -24, 2003.

[10] D. Garlan, S. Khersonsky, and J.S. Kim, “Model Checking Publish-Subscribe Systems”, *Proc. of SPIN 03*, Portland, Oregon, 2003.

[11] David Garlan and Mary Shaw, “An Introduction to Software Architecture”, *CMU-CS-94-166*, School of Computer Science, Carnegie Mello University, 1994.

[12] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, “Formally Specifying and Analyzing Software Architectural Specifications Using SAM”, *Journal of Systems and Software*, vol.71, no.1-2, pp.11-29, 2004, 1994.

[13] G.J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, Sept. 2003.

[14] J. S. Kim, and D. Garlan, “Analyzing architectual styles”, *Journal of Systems and Software*, 83(2010), pp. 1216-1235, 2010.

[15] M. Köhler-Bußmeier, “A Survey of Elementary Object Systems”, *3rd International Workshop on Logics, Agents, and Mobility (LAM'10)*, vol. 7, pp. 19-36, 2012.

[16] N. Medvidovic, R. Taylor, 2000. “A classification and comparison framework for software architecture description languages”. *IEEE TSE* 26 (1), 70–93, 2000.

[17] R. Milner, *Communicating and Mobile Systems: The Pi Calculus*, Cambridge University Press, June 1999.

[18] T. Murata, “Petri Nets: Properties, Analysis and Applications”. In *Proceedings of the IEEE*, vol.77, no.4, (1989) pp. 541-580.

[19] A. Sanchez , A. Madeira , L. S. Barbosa, “On the verification of architectural reconfigurations”, *Computer Languages, Systems and Structures*, v.44 n.PC, p.218-237, 2015.

[20] A. Saadi , M. Oussalah , A. Henni , D. Bennouar, “Handling the Dynamic Reconfiguration of Software Architectures using Intelligent Agents”, *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, pp.1-5, 2015.

[21] D. Xu, D., K. E. Nygard, “Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets”. *IEEE TSE*. 32(4), 265–278, 2006.

[22] Xu, D., Yin, J., Deng, Y. and Ding, J., A Formal Architecture Model for Logical Agent Mobility. *IEEE Trans. on Software Engineering*. vol. 29, no. 1 (2003), pp. 31-45.