Modeling and Analyzing Publish-Subscribe Architecture using Petri Nets

Junhua Ding[1, 2]
1) Dept. of Computer Science
East Carolina University
Greenville, NC 27587
dingj@ecu.edu

Dongmei Zhang[2]
2) School of Computer Sciences
China University of Geosciences
Wuhan, Hubei, China
jjielee@163.com

Abstract — **Software architecture is the foundation for the development of software systems. Its correctness is important to the quality of the software systems that have been developed based on it. Formally modeling and analyzing software architecture is an effective way to ensure the correctness of software architecture. However, how to effectively verify software architecture and use the results from formal modeling and analysis is important to the application of the approach. In this paper, software architecture is modelled using high level Petri nets, and the model is then checked with a model based testing tool called MISTA, and bounded model checking tool Alloy to ensure the correctness of the model. The approach is designed as a two-phase process consisting of model-based testing and bounded model checking to ensure it is both practical and rigorous for analyzing software architecture. We illustrated the idea and procedure via modeling and analyzing the Publish-Subscribe architecture. The result has shown that combining bounded model checking with model based testing is an effective extension to ensure the development quality.**

Keywords- software architecture; Petri net; model checking; model based software testing; publish-subscribe architecture

I. Introduction

Software architecture is an overall structure of a software system, which consists of a group of components and the connections among components in addition to the constraints applying to the connections. It is the foundation of product lines and many software systems were developed based on it. Therefore, correctness of software architecture is important to the quality of software systems that have been built on it. Formal modeling and analysis of software architecture offers a rigorous way to ensure the correctness of software architecture, which has been discussed in many articles [5]. However, results of formal modeling and analysis are difficult to be directly used for analyzing software implementation that was built based on the formal models due to the specification gap between models and their implementations. For example, if a model is specified using Petri nets, and the implementation language is Java, then the model checking results (*e.g.*, counter examples) of the Petri nets model cannot be directly used for testing the Java program. But model checking a complex Java application is infeasible and testing is still the practical way for program verification. Model-based software testing is an approach to bridge the gap between testing of a software model and its implementation, where models are used for guiding the test generation. In some cases, model level tests are first generated, and then they are transformed into program level tests. MISTA [17] is a model based software testing tool, which models a software system in high level Petri nets, and then the Petri net model is analyzed with simulation and model checking. Model level tests can be automatically generated according to selected test coverage criteria, and then these tests are automatically transformed into program level tests with help from mapping files. The program level tests can be directly used for testing the implementation. However, due to the grand challenge of modeling of a high level Petri net, model checking capability in MISTA is limited. In this paper, we extended MISTA with bounded model checking for analyzing Petri nets. Alloy analyzer is a bounded model checker for analyzing models specifying in Alloy language, which is a formal specification language based on first order relational logic [3][10]. Alloy analyzer is a constraint solver for automatically checking an Alloy model that specifies the structural constraints and behaviors of a software system [3]. Alloy finds all model instances for satisfying a checked property within the bounded scope, and it provides a visualization tool to illustrate all instances. Comparing the graphic instance to the corresponding Petri net model will be useful to better understand the Petri net model and create a better model. In addition, the instances are also useful for creating tests for testing interesting properties in the Petri net.

Publish-Subscribe (pub-sub) architecture is a well adopted event-based software architecture. The pub-sub architecture includes one or more components that publish events, and one or more components that subscribe them. The loose coupling of publish and subscribe components offers the flexibility of updating components and events in a system, but it also brings the complexity of analysis due to the large number of possibility of combination of event transferring scenarios [6]. Several analysis approaches such as model checking [6][8] have been attempted for analyzing pub-sub models. In this paper, we introduce Alloy into MISTA for analyzing Petri nets. First, a Petri net model is modeled and simulated, and then simple properties are verified using MISTA. After that, the Petri net model is converted into an Alloy model, which will be analyzed using Alloy analyzer. The analysis results can be used for improving the Petri net model and guiding generating tests for interesting properties. The analysis process is illustrated through modeling a general model of the pub-sub architecture in Petri nets. The general model can be easily extended for different versions of the pub-sub architecture. Based on the Petri net model, the pub-sub architecture was modelled in Alloy, and analyzed for interesting properties using Alloy analyzer. A Petri net model can be automatically transformed into an Alloy model.

DOI reference number: 10.18293/SKE2015-232

The main contribution of this paper is due to a two-phase rigorous and practically useful approach for analyzing software architecture. Since software architecture is the foundation for the implementation of many software systems, it is important to provide an easy-to-use technique such as simulation and testing for analyzing software architecture when they are still in the early development phase. But simulation and testing is not enough to ensure the correctness of important properties in software architecture. Rigorously checking the architectural model is necessary for ensuring the quality of the architecture especially in the later modeling phase. In our approach, the model-based testing assists ones to understand the modeling of software architecture, to check simple assertions and to test special scenarios for building a correct software architecture. In addition, model checking ensures the correctness of important properties modeled in the architectural model. The bounded model checker Alloy was smoothly extended to model based testing tool MISTA for enhancing the features in MISTA. Modeling and analyzing the pub-sub architecture is used to explain the idea and process, and to show the effectiveness of the proposed approach.

The rest of this paper is organized as follows: Section 2 presents a brief introduction to Alloy, PrT nets, model-based software testing and its tool MISTA. Section 3 introduces the proposed model based testing of the pub-sub architecture in Petri nets using MISTA. Section 4 discusses how to extend Alloy into MISTA, model and analyze the pub-sub architecture using Alloy, and how to convert a Petri net into an Alloy model. Section 5 reviews the related work, and section 6 concludes this paper.

II. BACKGROUND

A. Alloy

Each Alloy model is specified in Alloy language to define how to check the occurrence of a state change [3]. Each model represents a set of model instances, and Alloy analyzer is used to search for instances or counterexamples of a model. Alloy analyzer is a bounded model checker for analyzing a model within a finite scope a user specifies [3]. The analysis is sound and complete within the scope so that it never misses a counterexample within the scope. Alloy analyzer either finds a solution that satisfies a predicate defined in the model, or a counterexample that violates a given assertion [11]. An Alloy model includes a number of signatures and facts. A signature defines a set and a group of atoms associating with the set, and a fact defining a constraint that is assumed always to hold in the model. Analysis of a model is conducted for checking a predicate or an assertion of the model. The details of Alloy analyzer and language can be found in the project website [3] and the book [10]. Fig. 1 is an Alloy model (modified based on the original model described in the tutorial of Alloy [3]) for defining a simple file system, which includes files, directories and root. *sig FSObj* defines objects in a file system, *sig File*, *sig Dir* and *sig Root* define files and directories, which are also objects. The three facts define the global constraints that are: each directory is the parent of its contents, each object is either a file or directory, and a root does not have a parent. The *assert* declares that a path is acyclic, and check it for scope of 5 [3].

```
module fileSys
    abstract sig FSObj { parent: lone Dir}
    sig Dir extends FSObj { contents: set FSObj}
    sig File extends FSObj { }
    one sig Root extends Dir { } { no parent }
    fact {all d: Dir, o: d.contents | o.parent = d}
    fact { File + Dir = FSObj}
    fact { FSObj in Root.*contents}
    assert acyclic {no d: Dir | d in d.^contents}
check acyclic for 5
```

Figure 1. A sample Alloy model

B. PrT Nets

Predicate/Transition (PrT) nets are a high level Petri net for specifying concurrent systems. The definition of PrT nets used in this paper is same as the one defined in [18].

Definition 1 (PrT net) A PrT net is a tuple (*P, T, F, Σ, L, φ, M_0*), where: *P* is a finite set of predicates (first order places), *T* is a finite set of transitions and *F* is a flow relation. (*P, T, F*) forms a directed net. *Σ* is a structure consisting of sorts of individuals (constants) together with operations and relations. *L* is a labeling function on arcs. *φ* is a mapping from a set of inscription formulae to transitions, and M_0 is the initial or current marking.

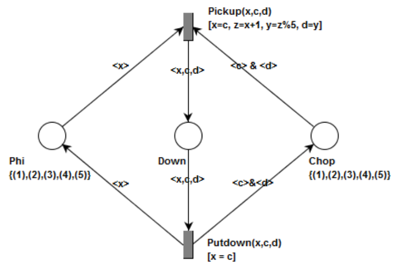

Figure 2. A PrT nets model for dining philosophers

Fig. 2 shows a simplified PrT net model for 5 dining philosophers' problem. 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* include tokens that are nature numbers representing philosophers or chopsticks, and each token in place *Down* represents a philosopher and his/her two chopsticks. Transition *Pickup* has two input places *Phi* and *Chop*, and one output place *Down*. The guard condition in transition *Pickup* is defined based on the relation between the tokens in place *Phi* and *Chop*: $x=c\&\&d=(x+1)\%5$, representing that a philosopher must get both of his or her left and right chopstick before he or she can eat (*pickup*) The guard condition in transition *Putdown* is defined based on the relation between the tokens in place *Phi* and *Chop*: $x=c$, representing a philosopher puts down chopsticks at both left side and right side.

C. Model-based Testing and MISTA

MISTA [14][17] is a model-based testing tool for automated generation and execution of tests. It generates tests in model level first and then program level tests are produced through transforming the one at model level. It specifies models in function nets, which is a type of PrT nets extended with inhibitor arcs and reset arcs [18]. It also provides a language for mapping the elements in function nets to implementation constructs so that it is possible to transform the model level tests into program level tests that can be executed against the system under test. In addition to test generation, MISTA includes simulation and limited model checking functions. It supports the step by step execution and random execution of a function net, and the execution sequences and token changing in each place are visualized for inspection. The test generator generates adequate model level tests (*i.e*., firing sequences of a function net) according to a selected coverage criterion such as reachability coverage, transition coverage, state coverage, depth coverage, and goal coverage. Test code generator generates test code in a target program language like Java or C++ from a given transition tree [17].

III. Modeling and Testing Software Architecture using PrT Nets

In this section, we are going to discuss an approach for analyzing a PrT net model using model-based software testing technique. In order to illustrate the basic idea and the process of the two-phase analysis approach, we model and analyze a pub-sub model using MISTA in this section and Alloy in next section.

A. *Modeling the Pub-Sub Architecture*

The pub-sub architecture is an event-based architecture, which includes one or more publishers that publish contents, and one or more subscribers that consume the contents. The publisher sends its contents as event messages through an event bus, and a subscriber subscribes its contents through an event message classification mechanism that classifies contents as channels [1].

In the PrT net model shown in Fig. 3, a publisher publishes its content as a message *msg* through transition *pub*, and the published content is notified to subscribers via transition *notify*, which models the event bus, and messages are classified as channels and stored in place *channel* by transition *classify*. A subscriber subscribes a channel message via transition *sub*, and the subscribed channel message is sent to the subscriber by transition *classify* when the message is available.

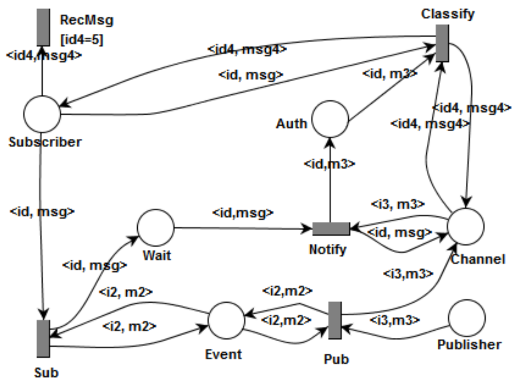

Figure 3. A PrT net model for the pub-sub architecture

B. *Testing the Pub-Sub Architecture*

As soon as a PrT net model is created and successfully compiled in MISTA, run it with random inputs to help developers to understand the model and detect easily found problems. If the simulation result is acceptable, verification of the goal reachability, assertions and deadlock states is conducted. After that, a set of tests can be generated based on selected coverage criteria, and these tests will be converted into program level tests for testing the corresponding implementation.

First, execute the PrT net model for the pub-sub with valid initial markings to simulate normal running scenarios of the model. For example, check a normal scenario that a publisher publishes a message, which is the type of messages that a subscriber has subscribed. It is important to check that the message is successfully classified and stored in the channel and the subscriber is notified, and finally the message is delivered to the subscriber. An example of the initial marking for checking above scenario in the PrT net in Fig. 3 is:

INIT Event(1,"1"), Publisher(1,"2"), Publisher(1,"1"), Subscriber(11, "0"), Subscriber(11, "s")

Second, verify the reachability of goal states and transitions, assertions and deadlock states in the PrT net for the pub-sub using the model checking capability in MISTA. When the reachability of all transitions of the PrT net in Fig. 3 was checked, transition *RecMsg* was unreachable was found since no any message with *ID*=5 was ever sent from any publisher. If a token such as ("5, "2") for place *Publisher* is added to the initial marking, all transitions will be reachable. Given a goal state such as *GOAL Subscriber(5, "2")*, then MISTA will find that the state is reachable. The PrT net model has termination states because published messages are delivered to subscribers and removed from their channels and it is possible that any channel has a message. The model has to be updated if a subscriber only receive copies of its subscribed messages and all messages will stay in the channel for a period of time.

Third, generate adequate tests for selected test coverage criteria using MISTA. For the PrT net model defined in Fig. 3, generate model level tests covering all states, all executable paths, goal states, and others. Since complex scenarios are not feasible to be checked with simulation or the limited model checking in MISTA, these complex scenarios shall be rigorously tested with model level tests thanks to the executable capacity of PrT nets. The model level tests are also used for generating program level tests via mapping the model to its corresponding programs. The program level tests will be used for testing the implementation of the model.

Although the PrT net model was checked with simulation of execution scenarios, verification of important properties, and testing with adequate tests for selected coverage criteria, the model is not guaranteed to be correct. Formal verification of a PrT net model is very difficult, but the analysis process of bounded model checking is fully automatic and it can guarantee the correctness of the checked properties within specified scope. Therefore, bounded model checker Alloy was chosen as an addition to MISTA to further analyze a PrT net model.

IV. Analyzing Software Architecture using Alloy

In this section, first we discuss how to model and analyze the pub-sub architecture in Alloy, and then we introduce a procedure for transferring a PrT net into an Alloy model.

A. *Analysis of the Pub-Sub Architecture*

The architecture defined in this section is specified following Acme style [1], which defines software architecture as a group of components and the connection for connecting the components via interfaces in addition to the constraints applying to the connection. The Alloy model of the pub-sub is defined in a hierarchical structure. The basic elements of the general software architecture are defined in an Alloy model serving as the foundation for modeling specific software architecture, and then a model of event based architecture is defined. The foundation model and the model of event based architecture were created based on those models introduced in [11]. The model for the pub-sub architecture was developed through extending above two models. In the foundation model, common software architecture elements such as components, connectors, ports and roles are defined as signatures, and basic constraints of software architecture are defined as facts. The model also defines a group of built-in functions and predicates for checking specific properties or looking for counterexamples. The pub-sub architecture is an event-based architecture that consists of loosely-coupled components that produce and consume events through the ports. In the model of event-based architecture, the signatures include two components: *AnnounceComponent*, and *ReceiveComponent,* for modeling an announce component and a receive component, respectively. Each component includes a set of ports as interfaces; one connector *EventConnector*, models the connector for connecting between roles and ports. Each connector includes two sets of roles as interfaces, two types of roles *AnnoucerRole* and *ReceiverRole*, and two types of ports *AnnoucePort* and *ReceivePort*. The connection between components is implemented through connecting ports to the corresponding roles. The pub-sub architecture is an extension of the event-based architecture with the subscriber selects messages through a message classification mechanism called channel. The following Alloy code is partial of the Alloy model for the pub-sub architecture, and the model was developed based on the Alloy models described in [11]. Channel is defined as a component, and a channel connector is defined for connecting a channel and its subscribers.

```
   sig PubPort extends AnnouncePort {}
   sig SubPort extends ReceivePort {}
   sig PubRole extends AnnouncerRole {}
   sig SubRole extends ReceiverRole {}
   sig PubComp extends Component{pubPort: PubPort}
   sig SubComp extends Component {subPort: SubPort}
   sig Channel extends Component {channelPort1:
SubPort, channelPort2: SubPort}
   sig EventBusConn extends EventConn {pubRole:
PubRole, subRole: SubRole}
   sig ChannelConn extends Connector {subRole:
SubRole, channelRole: ReceiverRole}
```

One fact described as follows is defined to ensure that each component or connector has appropriate number of roles or ports.

```
fact{
 (PubComp<:pubPort) in ports
 (SubComp<:subscribePort) in ports
 (Channel<:channelPort1) in ports
 (ChannelConn<:channelRole+ChannelConn<:subRole)in roles
 (EventBusConn<:pubRole+EventBusConn<:subRole)in roles
}
```

Based on above model, one can check interesting properties of the model, such as an architectural configuration or a set of constraints. The following predicate defines a set of constraints that define the mapping between ports and roles, and the essential elements in the pub-sub architecture. In the code, *self* is a term defined in Acme as a signature for extending *System*, and *all* and *some* represent universal and existential quantification, respectively.

```
abstract sig System {components: set Component,
connectors: set Connector}
one sig self extends System {}

pred pubsub_constraints(){
 some c:self.components|declaresType[c,PubComp]
 some k:self.components|declaresType[k,SubComp]
 some m:self.components|declaresType[m,Channel]
 some n:self.connectors|declaresType[n,EventBusConn]
 some f:self.connectors|declaresType[f,ChannelConn]
 all self:PubPort|
  (all r: self.~attachment|declaresType[r, PubRole])
 all self: SubPort|
  (all r:self.~attachment|declaresType[r, SubRole])
 all self:PubRole |
  (#(self.attachment)=1)&&
  (all p: self.attachment|declaresType[p, PubPort])
 all self:SubRole |
  (#(self.attachment)=1)&&(all p: self.attachment|
declaresType[p, SubPort])
 all self:EventBusConn |
  (some e:self.pubRole|declaresType[e, PubRole] &&
   some f:self.subRole|declaresType[f, SubRole])
 all self:ChannelConn|
  (some e:self.channelRole|declaresType[e, ReceiverRole]
&& some f:self.subRole|declaresType[f, SubRole])
}
```

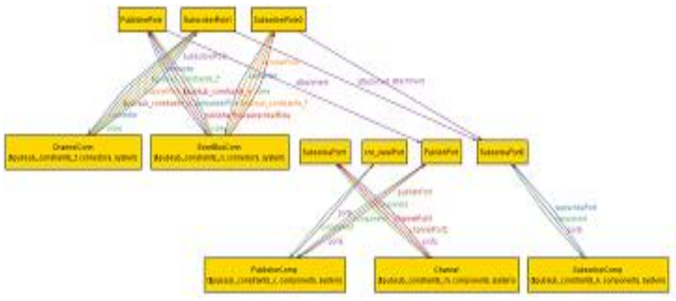

Figure 4. A snapshot of an instance for pub-sub architecture

Checking the predicate with a specified scope such as 5 using Alloy analyzer shall find instances that satisfy the predicate in the pub-sub model. The instances are helpful to understand the architecture and create better PrT net model. Fig. 4 shows a snapshot of an instance. One also can check whether a particular instance of pub-sub can be found or not in the model such as the following configuration of a pub-sub can be found with scope of 4. The configuration defines a simple pub-sub instance which consists of one subscriber, one publisher, one channel, one event bus connection, and one channel connection.

```
 pred Config_0(s:SubComp,e:EventBusConn,c:Channel,
              cc:ChannelConn, p:PubComp){
   attached[s.subscribePort, cc.subscriberRole]
   attached[cc.channelRole, c.channelPort2]
   attached[c.channelPort1, e.subscriberRole]
   attached[e.publisherRole, p.publishPort]
}
```

B. Translating a PrT Net into an Alloy Model

The Alloy model discussed in previous section was built directly according to the pub-sub architecture defined in Acme [1] for explaining the analysis process in a better way. In this section, we discuss how to translate a PrT net into an Alloy model to ensure the consistency between a PrT net and its corresponding Alloy model. Since Alloy models are declarative for defining how to recognize something has happened, and PrT net models are operational for defining how something can be accomplished [3]. It is fairly challenge to automate the transformation between a PrT net model and an Alloy model. In this section, we introduce a basic structure of the transformation using the dinning philosopher problem defined in Fig. 2 as an example. The idea of the transformation was developed based on the one for translating a regular place transition Petri nets into an Alloy model discussed in [16], which was extended for high-level Petri nets (*i.e.* PrT nets).

First, translate the basic elements (*i.e.* places, transitions, arcs and tokens) and basic constraints in a PrT net into signatures in Alloy.

```
   abstract sig Node{flow: set Node}
   abstract sig Token{}
   abstract sig Place extends Node{tokens:set
Token}{#tokens >= 0}
   abstract sig Transition extends Node{inp:set
Place, outp:set Place}
   abstract sig Arc{place:Place,tran:Transition}
```

Define common constraints as facts, such as a net consists of places and transitions, and a flow relation is only applied to a place to a transition or a transition to a place:

```
   fact {Node = Place + Transition}
   fact {all p: Place | p.flow & Place = none}
   fact {all t: Transition|t.flow&Transition=none}
```

Since each place has its own type of tokens, it is necessary to define a type of tokens for each place and define each place with its tokens. For example, we defined three types of tokens for the dinning philosopher problem.

```
 sig Eating extends Token{phi:set Int,left:set
Int,right:set Int}
 sig Chop extends Token{left:set Int,right:set Int}
 sig Ph extends Token{phi: set Int}
```

And then each place in the PrT net is defined as a signature with its tokens and each transition in the PrT net is also defined as a signature. The constraints are defined by the number of outward flows of each place or transition in the PrT net:

```
   sig phP extends Place{token: Ph}{#flow = 1}
   sig chopP extends Place{token:Chop}{#flow=1}
   sig eatP extends Place{token:Eating}{#flow= 1}
   sig Pick extends Transition {}{#flow = 1}
   sig Down extends Transition {}{#flow = 2}
```

Now, the structure of the net is defined as facts in Alloy based on the flow relations in the PrT net. The following Alloy code defines the connections between places and transitions of the PrT net in Fig. 2.

```
   fact {all p: phP|one t: Pick|t in p.flow}
   fact {all t: Pick|one p: phP|p in t.~(flow}
   fact {all p: phP|one t: Down|t in p.flow}
   fact {all t: Down|one p: phP|p in t.~(flow}
   fact {all p: chopP|one t: Pick|t in p.flow}
   fact {all t: Pick|one p: chopP|p in t.~(flow}
   fact {all p: chopP|one t: Down|t in p.flow}
   fact {all t: Down|one p: chopP|p in t.~(flow)}
   ……
   fact {all t: Pick|one p:chopP|p in t.flow}
   fact {all p: chopP|one t:Pick|t in p.~(flow)}
   fact {all t: Pick|one p:phP|p in t.flow }
   fact {all p: phP|one t: Pick|t in p.~(flow)}
   fact {all t: Pick|one p: eatP|p in t.flow }
   fact {all p: eatP|one t: Pick|t in p.~(flow)}
   ……
```

Finally, translate the fire conditions of all transitions as a predicate. The constraints for each transition include a pre-condition that is defined based the flow relation of the transition and the guard conditions of the transition, support functions, and a post-condition to define the effect of the firing.

```
-- pre-condition of transition Pick
   pick in c.flow and pick in p.flow
   and #(pick.~(flow)) = 2
   and e in pick.flow and #pick.flow = 1
   and #p.token.phi>0 and #c.token.right > 0
   and #c.token.left > 0
   and #(p.token.phi&c.token.right)>0
   and some x:Int in c.token.right and some y:Int
in c.token.left and x = mod(y,5)
        ……
-- post-condition of transition Pick
   some x: Int in c.token.right and some y:Int in
c.token.left
   and p.token.phi = p.token.phi - x
   and c.token.left = c.token.left - y
   and c.token.right = c.token.right - x
   and e.token.phi = e.token.phi + x
   and e.token.left = e.token.left + y
   and e.token.right = e.token.right + x

-- pre and post-condition of transition Down
   ……
```

Important properties such as safe or reachability can be defined as assertions to be analyzed by Alloy analyzer. Check the predicate *fire* with specified scope such as 6, Alloy analyzer will find instances for the model, which can be used for checking the original PrT net.

V. RELATED WORK

Software architecture has become an essential part in almost every phase of software development lifecycle [5]. Therefore, many researchers have proposed approaches and built tools for modeling and analyzing software architecture. In order to improve rigorousness of the analysis of software architecture and confidence of the quality of the architectural model, a variety of formal modeling and analysis approaches and tools have been introduced during past two decades. Garlan [5] has summarized the representative results of formal modeling and analysis of software architecture. Allen and Garlan [2] described a formal basis for an architectural connection, which has become the one of the most important work on formal modeling of software architecture. Model checking has been reported for formally analyzing software architecture. In [7], He and *et. al*. reported approaches for formally analyzing Petri nets using model checking and formal proof techniques. Ding and He proposed an approach for modeling checking a type of high level Petri nets in [4]. Several other researchers defined an executable semantics for software architectural modeling and analysis through simulation and/or formal verification [13]. For example, formal specification language Rapide [12] supports simulation, Chemical Abstract Machine [9] and Wright [1] support limited formal verification. However, few work on testing of software architecture have been reported [15] due to its nature of informal and non-executable of architecture models in general. Zhu and He [19] proposed a methodology for testing design and architectural models in high level Petri nets. Model based testing uses results from testing architectural models for testing their corresponding implementations. For example, model level tests for testing an architecture are transformed into program level tests for testing its implementation [17][19]. The approach discussed in this paper is used for analyzing software architecture in Petri nets via naturally combing informal analysis techniques like software testing and formal analysis techniques like bounded model checking in two-phase analysis. The integration of bounded model checking with model based testing improves the rigorousness of model-based testing so that to improve the confidence of the correctness of important properties holding in the architecture. Although the pub-sub architecture has been widely implemented in many software systems, formal modeling and analyzing of its architecture is difficult. Garlan, Khersonsky and Kim [6] introduced a reusable generic framework for modeling and checking the model using model checker SMV. Kim and Garlan [11] also investigated how to analyze software architecture using Alloy. The approaches introduced in the two papers are similar to the approach of bounded model checking of architectural models discussed in this paper. The technique was also extended for model-based testing to improve the analysis performance and effectiveness.

VI. Summary and Future Work

In this paper, we presented an approach for modeling and analyzing software architecture through studying the pub-sub architecture. The approach is designed as a two-phase process to ensure it is both practical and rigorous for analyzing software architectures in PrT nets. In the first phase, a PrT net model is analyzed using model-based testing techniques including simulation, model checking and testing with tool MISTA. The bounded model checking is conducted by converting a PrT net into an Alloy model inputting to model analyzer Alloy. Then model-based test cases are generated from the checked model for selected test coverage criteria and finally they are converted into the program level tests for testing the corresponding implementation. Model-based software testing with MISTA has been introduced in other publications [14][17], but the focus of this paper is on how to integrate bounded model checking into the process. The process of modeling and analysis of software architecture was illustrated by modeling and analyzing the pub-sub architecture. The approach is useful for analyzing software architecture in general, and also provides a framework for modeling and analyzing variety versions of pub-sub architecture. We plan to develop a tool to automate the transformation from a PrT net to an Alloy model.

Acknowledgments

This research is supported in part by award #CNS-1262933 from the National Science Foundation. Junhua Ding's research was also partially supported by the guest professorship grant from school of computer sciences at China University of Geosciences.

References

[1] Acme, http://www.cs.cmu.edu/~acme/, last accessed on March 10, 2015.

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

[3] Alloy: http://alloy.mit.edu, last accessed on March 10, 2015.

[4] 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.

[5] 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.

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

[7] 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.

[8] P. Hens, M. Snoeck, G. Poels, D. B. Manu, "A petri net formalization of a publish-subscribe process system", FBE Research Report KBI_1114, K.U.Leuven - Faculty of Business and Economics, June 2011.

[9] P. Inverardi, A. Wolf. "Formal specification and analysis of software architectures using the chemical abstract machine model." IEEE TSE, 21 (4), 373–386, 1995.

[10] D. Jackson, "Software Abstractions: Logic, Language and Analysis", the MIT Press, 2012.

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

[12] D. C. Luckham, J. Kenney, et al. "Specification and analysis of system architecture using rapide." IEEE TSE 21 (4), 336–355, 1995.

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

[14] MISTA, http://cs.boisestate.edu/~dxu/research/MBT.html, last accesed on March 12, 2015.

[15] D. Richardson, A. Wolf. "Software testing at the architectural level." In: Proc. of the 2nd Intl. Soft. Architecture Workshop. pp. 68–71, 1996.

[16] J. A. Robles, G.A. Solano, "Modeling Petri nets using Alloy," TENCON 2012 - 2012 IEEE Region 10 Conference , vol., no., pp.1,6, 19-22 Nov. 2012.

[17] D. Xu, "A Tool for Automated Test Code Generation from High-Level Petri Nets". 32nd Int. Conf. on Apps. and Theory of Petri Nets , Newcastle, UK, June 20-24, 2011.

[18] 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.

[19] H. Zhu, and X. He, "A methodology of testing high-level petri nets". Journal of Information and Software Technology. v44, pp. 473-489, 2002.