Using Monterey Phoenix to Formalize and Verify System Architectures

Jiexin Zhang*, Yang Liu†, Mikhail Auguston‡, Jun Sun§ and Jin Song Dong*
*School of Computing, National University of Singapore
{jiexinzh,dongjs}@comp.nus.edu.sg
†Temasek Laboratories, National University of Singapore
tslliuya@nus.edu.sg
‡Department of Computer Science, Naval Postgraduate School
maugusto@nps.edu
§ISTD, Singapore University of Technology and Design
sunjun@sutd.edu.sg

***Abstract*—Modeling and analyzing high-level software architectures are useful in helping understand the system structures and facilitate the proper implementation of user requirements. Despite its importance in the software engineering practice, the lack of formal description and verification support hinder the development of quality architectural models. In this work, we develop an automatic approach for modeling software architectures and verifying a wide range of properties in PAT framework. Firstly, we define the formal syntax and operational semantics for Monterey Phoenix (MP) architecture description language. This language is capable of describing both static and dynamic system behaviors, as well as supporting different architecture composition and views. Secondly, a dedicated model checking module for MP is implemented in the PAT verification framework based on the proposed formalization. Finally, a case study about the Radar Weapon system is presented to evaluate the usability, effectiveness, and scalability of our approach.**

I. Overview

Software Architecture plays a vital role in the high level design of a software system. Analogy to civil engineering, it represents the fundamental structural and behavioral descriptions of the software system during the engineering process. Software Architecture specification has been widely used in many fields as it can assist users to get an intuitive understanding of the whole system on one hand and facilitate different groups to cooperate together by giving them divisions with guidelines and objectives on the other. The challenges in this field have been focused on how to model a system in its early design phase to reduce the occurrences of bugs and mistakes. The development cost can also be greatly reduced if bugs and mistakes can be found in high-level architecture designs. In order to solve the above problems, several related techniques can be referred to, such as, term rewriting systems, theorem provers, and model checkers. Among them, model checking is an effective and competitive technology which can verify the system against properties through exhaustive and automatical space exploration.

In the past decade, formal modeling techniques have been applied to software architecture designs [21], which aimed at achieving precise specification and rigorous verification of the intended structures and behaviors in the design. The advantage of such verifications is to determine whether a modeled structure can successfully satisfy a set of given properties derived from the requirements of a system. Furthermore, automated verification provides an efficient and effective means for checking the correctness of the architecture design. A considerable number of architecture description languages have been proposed in the past years, e.g., Wright [3], [2], Darwin [16], ACME [10], and CHAM [12], [9]. Wright, Darwin and ACME capture the properties and structures of systems by employing the composed components interacted through connectors, where CHAM models system architecture in terms of molecules and transformation rules. However, one drawback of many existing approaches in the field lies in the limited verification support to the software architecture models specified in those notations. For example, Wright is considered as the prominent language in modeling the component and connector structures. It makes explicit use of parameterizing the specific behaviors of a particular type. This language is partially encoded into the FDR model checker, where a subset of the language constructs and limited model checking properties such as compatibility checking and deadlock analysis are available. In the Darwin language, the system behaviors are specified by finite state process algebra. It can describe concurrent and distributed systems and has its own model checker LTSA [15] to perform verification. The language can handle reconfiguration behaviors well but cannot address the issues about the complex interactions among reconfiguration unites. In the case of ACME language, it is intended to support mapping from one architecture description language to a logical formalism and adopts an open semantic framework to reason the model. Kim and Garlan [13] proposed the modeling and verification of architecture styles using the Alloy language and its analyzer. In their approach, a few architecture styles based on ACME descriptions were translated and verified using Alloy. Although it offers a useful insight to the ability of applying Alloy in automating the verification of architecture descriptions, the performance issue

is a practical limitation of the research. The problem arisen from large scope architecture models dramatically expanding the search spaces of the verification in the Alloy SAT solver. To overcome this problem, Wong et al. [20] proposed a model splitting approach for the parallel verification of Alloy based architecture models using their underlying styles. The approach improved the performance of the verification, however, the overheads of the model decomposition as well as the dependency issues among the sub-models during the parallel verification phase still remain as challenges. The CHAM language has an effective way to express system properties but with less verification support.

In this paper, we present an automated approach to the modeling and verification of software architectures in the PAT framework [1] [14]. We proposed the syntax and operational semantics for a software architecture description language – Monterey Phoenix (MP) [4], [5], [6]. Via this language, the behavior of the system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of event trace is specified using event grammars and other constraints organized into schemas. The structure of the system can be clearly and precisely designed based on the behavior models which provides a topology representation of how the system is composed and operated to users. In this work, we extend MP with a rich set of syntax to describe concurrent communications between the components and connectors of the system. We formally define the syntax and operational semantics to provide the foundation of formal analysis. The MP language is capable of describing both static and dynamic system behaviors, as well as supporting different architecture composition and views. Based on the formal semantics, we further developed a dedicated model checking module in the PAT verification framework, which supports modeling, simulation and verification of software architecture models. Finally, we demonstrate our approach with the architecture modeling and verification of the Radar Weapon System, where the effectiveness and scalability of the approach are evaluated.

The rest of the paper is organized as follows. Section II introduces the basic concepts and language features of M-P language. Section III defines the syntax and operational semantics for MP in PAT framework. Section IV illustrates different properties we can verified based on MP models. Section V demonstrates the modeling and verification of a case study using the MP module in the PAT model checker with evaluation results. Section VI concludes the paper and discusses the future work.

II. Basic Concepts

In this section, we will introduce the basic concepts and language features of Monterey Phoenix language. The software architectures are specified based on behavior models. The behavior of a system is defined as a set of events (event trace) with two basic relations: *precedence* (PRECEDES) and *inclusion* (IN). In case of *precedence*, it means two events are ordered in time sequence, that is one event happens before the other event. In case of *inclusion*, it represents one event appears inside another event. Under this relation, events can be defined in an appropriate level of granularity and with hierarchical structures. The two basic relations define a partial order between events because two events may happen concurrently that they are not necessarily ordered. Both relations satisfy properties of transitive, non-commutative, non-reflexive, and distributivity.

A. Event Grammar

The structure of event trace is specified by the predefined event grammar rules. These rules specifie the structure of a particular event type in terms of PRECEDES and IN relations, which is expressed in a form of:

A :: right-hand-part;

There are the following event patterns for use in the right hand part of grammar rules. Where A, B, C, D stand for event type names or event patterns, events are visualized by small squares, the two basic relations are visualized by arrows. Among which, the PRECEDES relation is denoted by continues arrow and the IN relation is denoted by dotted arrow.

1) A : (B C D);

The first event pattern is sequence which represents ordering of events under the PRECEDES relation. This rule means a type A event contains ordered events b, c, and d, matching B, C, and D event type. That is event b, c, d are IN event a, event b PRECEDES event c, and event c PRECEDES event d. This grammar rule may contain a sequence of two or more events, like ”A:(B C);” or ”A:(B C D E);”. The Figure 1 depicts the event trace specified by this rule:

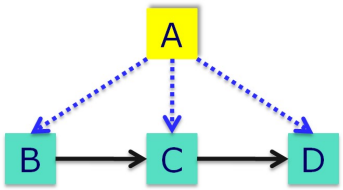

Fig. 1. Sequence Rule

2) A : (* B *);

The above rule denotes a set of zero or more events of type B with PRECEDES relation between them. All events of type B are IN event of type A. Users can set a particular scope for this rule in the following way: ”A: (* <startScope-endScope> B *);”, where ”startScope” and ”endScope” are nonnegative integers. A valid scenario for this rule is shown in Figure 2:

3) A : {B C D};

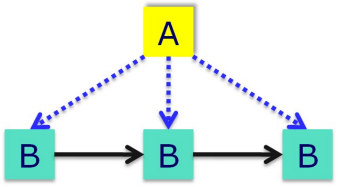

Fig. 2. Scope Sequence Rule

The third rule denotes a set of events B, C and D without PRECEDES relation between them. It represents an event a of the type A contains unordered events b, c and d belonged to event type B, C and D. The events b, c, d are all IN event a. This grammar rule may contain a sequence of two or more events, like "A:{B C};", "A:{B C D E};". The event trace in Figure 3 specifies a valid scenario for this rule:

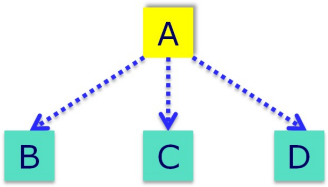

Fig. 3. Set Rule

4) A : {* B *};

The fourth rule denotes a set of zero or more events of type B without an ordering relation between them. Similar to scope sequence rule, users can set a particular scope for this rule in the following way: "A: {* <startScope-endScope> B *};", where "startScope" and "endScope" are nonnegative integers. A valid scenario for this rule is shown in Figure 4:

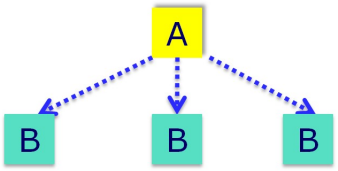

Fig. 4. Scope Set Rule

5) A : [B];

This rule denotes an optional event B, all valid scenario for this rule are presented in Figure 5:

Fig. 5. Option Rule

6) A : (B | C | D);

The sixth rule denotes an alternative - event A can include event B, event C or event D. All valid scenario for this rule are shown in Figure 6:

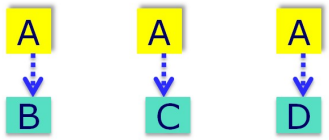

Fig. 6. Choice Rule

The behavior model of a software system is specified using a set of event traces together with some constraints. The basic concepts of MP is inspired by Z schema [17] and the architecture concept of *configuration*. For a traditional *configuration*, it usually contains a collection of components and connectors, where components are used to capture the behavior of each part of the system and connectors can specify the interactions among different components. In terms of MP model, both components and connectors can be expressed by *root* events, while other events are used to specify the interior event structures and interactions.

B. Share All

In addition to the basic grammar rules, MP also provides a mechanism for synchronizing different root events through specific predicate which is the *share all* constraint. This predicate plays a role similar to the parallel operation in CSP [11]. The following two items shows different structures of the share all constraint:

- A, B **SHARE ALL** D;
- A+B, C **SHARE ALL** D;

The events in the left side of the share all constraint should be root events only, therefore, event A, B, C are all root events. The above two items are both used to define the common events of multiple root events but with a subtle difference. The first one means root event A, B share all event D. So the occurrences of event D in event A is same as that in event B. The second constraint denotes that root event A with the addition of root event B share all event D with event C. The together times of event D occurring in event A and B are same as that in C. In the following, we will use an example of a simple client server architecture to illustrate this constraint.

ROOT *Server* : (∗ *Send* ∗);
ROOT *Client* : (∗ *Receive* ∗);
ROOT *Connector* : (∗ *Send Receive* ∗);
Server, *Connector* **SHARE ALL** *Send*;
Client, *Connector* **SHARE ALL** *Receive*;

Fig. 7. MP Codes for Client Sever Structure

This structure requires the client and server components involved in a strictly synchronized communication. Each *Send* event can only appears when the previous *Receive* event has

been accomplished. A valid transaction of this structure is shown in Figure 8 in the case of scope 2.

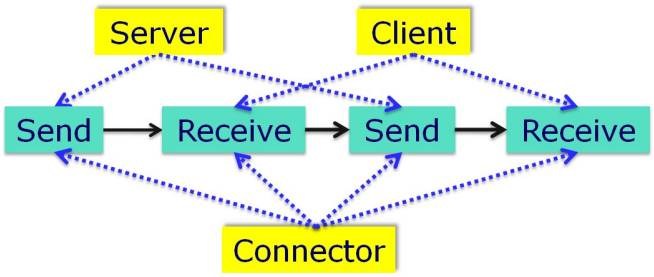

Fig. 8. Client Server Structure

C. Slice

For the assertion language, MP proposes a useful clause *Slice* to represents a set of concurrent events with in the event trace. The => operator here is the PRECEDES relation.

$$Concurrent(x, y) \equiv \neg(x => y) \wedge \neg(y => x)$$

Slice is a set of events from the event trace, such that

$$\forall x, y \in Slice \quad Concurrent(x, y)$$

Slice can be viewed as a special relation used in assertions which complements the PRECEDES and IN relations on the function.

III. Formal Syntax and Semantics

After introducing the basic concepts of MP language, we will present the formal syntax and operational semantics defined for it in PAT framework.

A. Syntax of MP

The model described by MP language is driven by event in the underlying execution. Events in MP can be classified into three categories according to the language features. They are root event, leaf event and middle event respectively. Among them, root event and middle event can be viewed as pattern-list references; leaf event is the atomic event which is executed at each step. The MP model is organized by parallel all the root events and execute them concurrently. The syntax of MP model is defined as below:

$$model\ M ::= P_1 \,||\, P_2 \,||\, ... \,||\, P_n \qquad \text{– model of MP}$$

where the P_i denotes the ith root event. In MP model, multiple root events can have several leaf events in common. The common leaf events are provided by SHARE ALL constraints. All the root events should execute their common leaf events simultaneously. For the respective other events, they can execute them in interleave order. The behavioral aspects of the non-leaf events is called as pattern-list which is the key part of MP. The syntax of pattern-list is:

$$\begin{array}{lll}
P ::= & e_l/program/ & \text{– atomic leaf event} \\
| & P\ Q & \text{– sequence} \\
| & (P_1 \mid P_2 \mid .... \mid P_n) & \text{– choice} \\
| & e_{nl} : P & \text{– nested event} \\
| & \{P_1, P_2, ...P_n\} & \text{– set concurrency} \\
| & \{* < a_1 - a_2 > P*\} & \text{– scope set concurrency} \\
| & (* < a_1 - a_2 > P*) & \text{– scope sequence} \\
| & P\ when\ (Q_1, \cdots, Q_n) & \text{– when} \\
| & if\ b\ \{P\}\ else\ \{Q\} & \text{– conditional choice} \\
| & while\ b\ \{P\} & \text{– while loop} \\
| & ref\{p\} & \text{– pattern reference}
\end{array}$$

where e_l denotes the atomic leaf event which can have a paragraph of program attached with; e_{nl} is non-leaf event that includes both root event and middle event. The capital letters P and Q are used to denote pattern-lists. We should define the scope for the set scope and sequence scope operations in order to make the event trace finite. Therefore, a_1 and a_2 are two nonnegative integers that define the lower bound and upper bound of the scopes.

A leaf event e_l can execute individually or execute together with a paragraph of program if attached with. $P\ Q$ behaves as pattern-list P first until its termination and then behaves as pattern-list Q. The Choice $(P_1 \mid P_2 \mid .... \mid P_n)$ is made internally and non-deterministically where any P_i may execute subsequently. The nested event defines the inclusion relation inside the pattern-list which performs as P. The set concurrency $\{P_1, P_2, ...P_n\}$ denotes interleaving execution where any P_i may perform their local actions without referring to each other. The scope set concurrency $\{* < a_1 - a_2 > P*\}$ defines several same pattern-list P execute without PRECEDES order. In contrast, scope sequence $(* < a_1 - a_2 > P*)$ defines several same pattern-list P execute in PRECEDES order. Each pattern-list P must performs after the previous one has been finished. The number of pattern-list P must fall in the predefined scope in both of the above two operations. The *When* structure is similar to the interrupt operation in CSP or the exception handling construct in traditional programming languages such as C# and JAVA. When the system execute pattern list P, either initial event of pattern list Q_i can be triggered and interrupt the normal execution. After that, the system control will be switched to pattern-list Q_i. The user can also attach a *RESTART* clause after the when construct which can resume the trace from the beginning of the pattern-list after the interruption. The $if\ b\ \{P\}\ else\{Q\}$ is a conditional branching, when b is evaluated to be true, the system performs P, else performs Q. Similarly, according to the boolean value of expression b, the while loop behaves continuously as pattern-list P or finishes the while loop immediately. A pattern-list expression could be given a name for referencing in a system model.

The Figure 9 presents a simple example of an ATM

SCHEMA *ATM Withdraw*
ROOT *Customer* : ($*$ *insert_card* ((*identification_succeeds request_withdrawal* (*get_money* | *not_sufficient_funds*))
| *identification_fails*) $*$);
ROOT *ATM system* : ($*$ *read_card validate_id* (*id_successful check_balance* (*sufficient_balance dispense_money*
| *insufficient_balance*) | *id_failed*) $*$);
ROOT *Data_Base* : ($*$ (*validate_id* | *check_balance*) $*$);
Data_Base, *ATM_system* **SHARE ALL** *validate_id*, *check_balance*;

Fig. 9. An ATM Withdraw Example

Withdraw system, which demonstrates how to integrate the behavior of environment with the behavior of system. The ATM_system and Data_Base synchronized through the valid_id and check_balance events. Each event trace generated through this schema represents a valid user case example.

B. Operational semantics

After giving the syntax, in this section, we will present the operational semantics, which translates a model into a Labeled Transition System (LTS). The sets of behaviors can be extracted from the operational semantics thanks to congruence theorems. In order to define the operational semantics, we first define the notion of system configuration to capture the global system state during system executions. A MP *system state* is presented by a pair of value (V,P), where V is a function mapping a variable name to its value, which we refer to as a valuation function, and P is an event pattern-list. The operational semantics for pattern-list is presented as firing rules associated with each pattern-list construct. Let Σ denote a set of events. For simplicity, a function $upd(V,prog)$, to which given a sequential *prog* and V, returns the modified valuation function V' according to the semantics of the program. We write $V \vDash b$ (or $V \nvDash b$) to denote that condition b evaluates to be true (or false) given V.

Figure 10 illustrates the firing rules. In *event* rule, the model behaves as event e_l, and updates the values of global variables in the meanwhile. In *sequence* rules, the model executes pattern-list P first. When P is found to be finished, the model will continue to execute pattern-list Q. The *alternative* rule is a multiple choice rule. The system can choose any pattern-list to execute subsequently. The *nested event* rule means that you can place a pattern-list behind a middle event . The middle event can be regarded as a pattern-list reference. The *root* rules are used to organize the whole root events, so P and Q here are both root pattern-lists. The common events of P and Q are defined by SHARE ALL constraints. The *When* rule denotes a pattern-list which can be interrupted by multiple pattern-lists and can restart from the beginning. So it can perform in a recursive mode. The *iteration* rule describes a scope sequence operation which means the pattern-list P can happen sequentially for a number of times. a_1 and a_2 here are used to define the sequence scope. The *set* rule means that a set of various pattern-lists execute concurrently without an ordering relation between them. The $set-iteration$ rule denotes a number of pattern-list P which execute concurrently. The two *conditional* (con_1 and con_2) choice rules define how to execute the conditional choice. According to the boolean value of expression b, the model behaves either as pattern-list P or as pattern-list Q. Here, the expression b is composed by global variables. The two $while-loop$ ($while_1$ and $while_2$) rules define how the while loop works. According to the boolean value of expression b, the model behaves continuously as pattern-list P or finishes the while loop.

A MP model is expressed by a 3-tuple $(Var, init, P)$, where *Var* is the set of values of global variables, *init* is the initial value of variables, and P represents pattern-list which is got via parallel all the root events. The model is translated into the Label Transition System to perform simulation and verification. A Label Transition System(LTS) is represented by a 3-tuple M $(S, init, Tr)$ where S denotes the set of states, *init* denotes the initial state which belongs to S, Tr is the transition relation which has the form of (S, e, S') where e is an event; S and S' are system configurations before and after the transition.

IV. Verification

The module implemented in PAT to support MP language comes with user friendly interfaces, featured model editor and animated simulator. The user friendly simulator can interactively and visually simulates system behaviors by random simulation, user-guide step by step simulation, complete state graph generation and counterexample visualization. Most importantly, it implements various verification techniques catering for different properties including deadlock-freeness, reachability, Linear Temporal Logic(LTL) properties (with or without fairness assumptions).

There are two types of properties we are concerned with in model checking. One is the safety property which guarantees nothing bad happens. Another is the liveness property which alerts something good eventually happens. Both deadlock-freeness and reachability assertions adapts to the safety property checking. There are two searching strategies: depth-first-search (DFS) and Breadth-first-search (BFS) implemented in PAT to support the safety property

$$\frac{}{(V, e_l\ /prog/) \xrightarrow{e_l} (upd(V, prog), \epsilon)}\ [\ event\ ] \qquad \frac{(V,P) \xrightarrow{e_l} (V',P')}{(V, e_{nl} : P) \xrightarrow{e_l} (V',P')}\ [\ nested\ event\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P')}{(V,P\ Q) \xrightarrow{e_l} (V',P'\ Q)}\ [\ sequence1\ ] \qquad \frac{(V,Q) \xrightarrow{e_l} (V',Q'), P = \epsilon}{(V,P\ Q) \xrightarrow{e_l} (V',Q')}\ [\ sequence2\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P'), P\ Q\ NOT\ SHARE\ ALL\ e_l}{(V,P \mid\mid Q) \xrightarrow{e_l} (V',P' \mid\mid Q)}\ [\ root\ 1\ ] \qquad \frac{(V,Q) \xrightarrow{e_l} (V',Q'), P\ Q\ NOT\ SHARE\ ALL\ e_l}{(V,P \mid\mid Q) \xrightarrow{e_l} (V',P \mid\mid Q')}\ [\ root\ 2\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P'), (V,Q) \xrightarrow{e_l} (V',Q'), P\ Q\ SHARE\ ALL\ e_l}{(V,P \mid\mid Q) \xrightarrow{e_l} (V',P' \mid\mid Q')}\ [\ root\ 3\ ] \qquad \frac{1 \le i \le n, (V,P_i) \xrightarrow{e_l} (V',P_i')}{(V,(P_1 \mid P_2 \mid .... \mid P_n)) \xrightarrow{e_l} (V',P_i')}\ [\ alternative\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P')}{(V,P\ when\ (Q_1,\cdots,Q_n)) \xrightarrow{e_l} (V',P'\ when\ (Q_1,\cdots,Q_n))}\ [\ when1\ ] \qquad \frac{(V,Q_i) \xrightarrow{e_l} (V',Q_i')\ ,\ 1 \le i \le n}{(V,P\ when\ (Q_1,\cdots,Q_n)) \xrightarrow{e_l} (V',Q_i')}\ [\ when2\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P')\ ,\ 0 \leqslant a_1 \le a \le a_2\ ,\ a \neq 0}{(V,(* < a_1 - a_2 > P*)) \xrightarrow{e_l} (V',P'\ \underbrace{P\ \cdots\ P}_{a-1})}\ [\ iteration\ ] \qquad \frac{(V,P_i) \xrightarrow{e_l} (V',P_i')\ ,\ 1 \le i \le n}{(V,\{P_1,\cdots,P_i,\cdots,P_n\}) \xrightarrow{e_l} (V',\{P_1,\cdots,P_i',\cdots,P_n\})}\ [\ set\ ]$$

$$\frac{(V,P) \xrightarrow{e_l} (V',P')\ ,\ 0 \leqslant a_1 \le a \le a_2\ ,\ a \neq 0}{(V,\{* < a_1 - a_2 > P*\}) \xrightarrow{e_l} (V',\{P',\underbrace{P,\cdots,P}_{a-1}\})}\ [\ set - iteration\ ]$$

$$\frac{V \vDash b, (V,P) \xrightarrow{e_l} (V',P')}{(V,if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e_l} (V',P')}\ [\ con1\ ] \qquad \frac{V \nvDash b, (V,Q) \xrightarrow{e_l} (V',Q')}{(V,if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e_l} (V',Q')}\ [\ con2\ ]$$

$$\frac{V \vDash b, (V,P) \xrightarrow{e_l} (V',P')}{(V,while\ b\ \{P\}) \xrightarrow{e_l} (V',P'\ \ while\ b\ \{P\})}\ [\ while1\ ] \qquad \frac{V \nvDash b}{(V,while\ b\ \{P\}) \to (V,\epsilon)}\ [\ while2\ ]$$

Fig. 10. Monterey Phoenix Operational Semantics

checking. For liveness properties, Linear Temporal Logic is a good candidate to achieve the goal as the MP model makes explicit use of the events, states and variables. Furthermore, the Linear Temporal Logic provides a very intuitive but very mathematically precise notation for expressing properties about the LT relation between the states/events in execution [8]. Given the Client Server structure, we want to verify whether each *Send* event performed by server will be matched with a *Receive* event from client eventually. It could be stated as below, where $\Box$ and $\Diamond$ are modal operators which denote 'always' and 'eventually' respectively.

$$\Box\ (Send\ \Rightarrow\ \Diamond Receive)$$

Such properties are very important in demonstrating the normal operations of systems. The integrated LTL formula [19] is:.

$$\phi ::= p \mid a \mid \neg\,\phi \mid \phi \wedge \psi \mid X\phi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

where p ranges over a set of propositions (formulated via predicates on global variables in MP) and a ranges over the events. Let $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots \rangle$ be an infinite execution. Let π^i be the suffix of π starting from s_i.

$$\begin{array}{ll}
\pi^i \vDash p & \Leftrightarrow s_i \vDash p \\
\pi^i \vDash a & \Leftrightarrow e_{i-1} = a \\
\pi^i \vDash \neg\,\phi & \Leftrightarrow \neg(\pi^i \vDash \phi) \\
\pi^i \vDash \phi \wedge \psi & \Leftrightarrow \pi^i \vDash \phi \wedge \pi^i \vDash \psi \\
\pi^i \vDash X \wedge \phi & \Leftrightarrow \pi^{i+1} \vDash \phi \\
\pi^i \vDash \Box\phi & \Leftrightarrow \forall j \ge i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \Diamond\phi & \Leftrightarrow \exists j \ge i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \phi U \psi & \Leftrightarrow \exists j \ge i \bullet \pi^j \vDash \psi \wedge \\
 & \qquad \forall k \mid i \le k \le j-1 \bullet \pi^j \vDash \phi
\end{array}$$

A model satisfies ϕ if and only if every infinite trace satisfies ϕ. A variety of properties can be expressed in LTL formulaes very concisely. In PAT framework, users can verify LTL properties with or without fairness. There are different levels of fairness supported which includes global fairness, event-level strong and weak fairness. If readers are interested in this part may refer to [18] for details.

V. Case Study and Evaluation

A. Modeling and Verifying Radar Weapon System

In this section, we apply our approach to the Radar Weapon system [7] to demonstrate the MP language as well as the MP module implemented in PAT. This system is composed by five subsystems. They are Generator, Radar, Weapon and Enemy_missile. The Generator is in charge of supplying enough power for Radar and Weapon subsystems when both of them are deployed. The environment is represented by Enemy_missile which can hit any of Generator, Radar, or Weapon. The consequence is causing termination of energy production or consumption correspondingly. The Enemy_missile may be detected by Radar, which in turn activates the Weapon. The Weapon can hit or miss the Enemy_missile when deployed. If the Generator is hit by the Enemy_missile, the deployed Radar, Weapon will get in the critical state of missing the power supply. The Control subsystem is used to coordinate the behaviors of Generator and Radar. The detailed MP code of this system is displayed in Figure 11

A variety of interesting properties could be verified in terms of Radar Weapon system. Three of them are picked to do demonstration. The first one is the deadlock-free analysis. This property is checked to guarantee the system never reaches a deadlock state. The second property verifies that wether the system will always win eventually. Property three states that whenever the Generator is terminated the Radar will stop working eventually.

B. Performance Evaluation

The Process Analysis Toolkit (PAT) [14] is designed to apply state-of-the-art model checking techniques for system analysis. It is designed to be an extensible and modularized framework that allows users to build customized model checkers to support the analysis of different system notations. The MP module in PAT is dedicated to support the MP model analysis and verification. We conducted experiments on the Client Server, ATM Withdraw, and Radar Weapon systems to evaluate the performance.
Table 1 shows the experiment results. The data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 3GB memory. Symbol '-' denotes out of memory. To be added...

VI. Conclusion and Future Work

In this paper, we present an automated approach for the modeling and verification of MP models in the PAT framework. We first defined the formal syntax and operational semantics of the MP architecture specification language. This language is capable of describing both static and dynamic system behaviors, as well as supporting different architecture composition and views. Based on the formal semantic we implemented a dedicated model checking module for MP in the PAT verification framework. Finally, we demonstrate the effectiveness of our approach through a case study of a Radar Weapon System modeling and verification. In addition, performance evaluations are presented to measure the scalability of the approach.

In the future, we plan to extended MP language with real-time and probabilistic properties to capture the quantitative time and uncertainty factors of different components in a software system. We will also develop a Graphic User Interface (GUI) to assist the visual design of the software architectures in the PAT framework. The GUI should provide diagram representations of the architecture models as well as support the definitions of the formal specifications on the design. In addition, we can further our work via designing an architecture style library in PAT which embodies a set of commonly used architecture styles to facilitate the modeling process. Some hot architecture styles such as cloud computing and Service-oriented architectures can be included.

SCHEMA *RadarWeaponSystem*
// ========== *Model part*
ROOT *Generator* : $(*\ <1-1>$ *Idle Generator_On Generating Generator_Off* $*)$
WHEN { *Generator_hit* => *Generator_Off Repair* [*RESTART*] };
ROOT *Radar* : $(*\ <1-1>$ *Idle Radar_On Radar_Working Radar_Off* $*)$
WHEN { *Radar_hit* => *Radar_Off Repair* [*RESTART*] };
Radar_Working : $(*\ <1-1>$ (*Target_detected* | *No_target*) $*)$;
Target_detected : *Weapon_On*;
ROOT *Weapon* : $(*\ <1-1>$ (*Idle* | *Weapon_On Shoot Recharge*) $*)$
WHEN { *Weapon_hit* => *Repair* [*RESTART*] };
Shoot : (*Hit* | *Miss*);
ROOT *Control* : $(*\ <1-1>$ *Generator_On Radar_On Monitoring*
Radar_off Generator_Off $*)$
WHEN { *Generator_hit* => *Generator_Off Repair* [*RESTART*] ,
Radar_hit => *Radar_Off Repair Radar_On* [*RESTART*] };
ROOT *Enemy_missile* : $(*\ <1-1>$ (*Approaching* | *Target_detected*) $*)$ *Boom*
WHEN { *Hit* => *Win* };
Boom : (*Generator_hit* | *Radar_hit* | *Weapon_hit* | *Miss*);
// ========== *Constraint part*
Radar, *Enemy_missile* **share all** *Target_detected*;
Radar, *Weapon* **share all** *Weapon_On*;
Weapon, *Enemy_missile* **share all** *Hit*, *Weapon_hit*;
Control, *Generator* **share all** *Generator_On*, *Generator_Off*;
Control, *Radar* **share all** *Radar_On*, *Radar_Off*;
(*Generator* + *Radar*), *Control* **share all** *Repair*;
Control, *Generator*, *Enemy_missile* **share all** *Generator_hit*;
Control, *Radar*, *Enemy_missile* **share all** *Radar_hit*;
// ========== *Assertion part*
#**assert** *RadarWeaponSystem deadlockfree*;
#**assert** *RadarWeaponSystem* $\models \Box\Diamond$*Win*;
#**assert** *RadarWeaponSystem* $\models \Box($*Generator_Off* $\Rightarrow \Diamond$*Radar_Off*$)$;

Fig. 11. MP code of The Radar Weapon System

REFERENCES

[1] PAT: Process Analysis Toolkit. http://www.comp.nus.edu.sg/~pat/research/.

[2] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *FASE*, pages 21–37, 1998.

[3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[4] Mikhail Auguston. Monterey phoenix, or how to make software architecture executable. In *OOPSLA Companion*, pages 1031–1040, 2009.

[5] Mikhail Auguston. Software architecture built from behavior models. *ACM SIGSOFT Software Engineering Notes*, 34(5):1–15, 2009.

[6] Mikhail Auguston, James Bret Michael, and Man tak Shing. Environment behavior models for automation of testing and assessment of system safety. *Information & Software Technology*, 48(10):971–980, 2006.

[7] Mikhail Auguston and Clifford Whitcomb. System architecture specification based on behavior models. 2010.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] Flavio Corradini, Paola Inverardi, and Alexander L. Wolf. On relating functional specifications to architectural specifications: A case study. *Sci. Comput. Program.*, 59(3):171–208, 2006.

[10] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In *CASCON*, page 7, 1997.

[11] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, 1978.

[12] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Trans. Software Eng.*, 21(4):373–386, 1995.

[13] Jung Soo Kim and David Garlan. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ROSATEA '06, pages 70–80, New York, NY, USA, 2006. ACM.

[14] Yang Liu, Jun Sun, and Jin Song Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199, 2011.

[15] Jeff Magee. Behavioral analysis of software architectures using ltsa. In *ICSE*, pages 634–637, 1999.

[16] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT FSE*, pages 3–14, 1996.

[17] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[18] J. Sun, Y. Liu, J. S. Dong, and H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *ICFEM'08*, volume 5256 of *LNCS*, pages 318–337. Springer, 2008.

[19] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating specification and programs for system modeling and verification. In *TASE*, pages 127–135, 2009.

[20] Stephen Wong, Jing Sun, Ian Warren, and Jun Sun. A scalable approach to multi-style architectural modeling and verification. In *Proceedings of the 13th IEEE International Conference on on Engineering of Complex Computer Systems*, ICECCS '08, pages 25–34, Washington, DC, USA, 2008. IEEE Computer Society.

[21] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.