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
†School of Computer Engineering, Nanyang Technological University, Singapore
yangliu@ntu.edu.sg
‡Department of Computer Science, Naval Postgraduate School, Monterey, California, USA
maugusto@nps.edu
§ISTD, Singapore University of Technology and Design
sunjun@sutd.edu.sg

***Abstract*—Modeling and analyzing software architectures are useful for helping to understand the system structures and facilitate proper implementation of user requirements. Despite its importance in the software engineering practice, the lack of formal description and verification support hinders the development of quality architectural models. In this work, we develop an approach for modeling and verifying software architectures specified using Monterey Phoenix (MP) architecture description language. Firstly, we formalize the syntax and operational semantics for MP. This language is capable of modeling system and environment behaviors based on event traces, as well as supporting different architecture composition operations and views. Secondly, a dedicated model checker for MP is developed based on PAT verification framework. Finally, several case studies are presented to evaluate the usability and effectiveness of our approach.**

I. Overview

Software Architecture plays a vital role in the high level design of a software system. In analogy to the civil engineering, it represents the fundamental structural and behavioral descriptions of the software system during the engineering process. Software Architecture specifications have been widely used in many fields to assist users to get an intuitive understanding of the whole system on one hand, and to facilitate different groups to cooperate together by giving them 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. The development cost can be greatly reduced if mistakes are found early in high-level architecture designs. In order to address the above problems, the model checking technique has been widely used in this field. Model checking is an effective means to verify the system against properties through automatic and exhaustive search space exploration.

Among the various architecture description languages, Monterey Phoenix (MP) [4], [5], [6] is a precise and innovative language which describes the system behavior based on rigorous event grammar rules. In 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 grammar rules and other constraints organized into schemas. The structure of the system can be designed based on the behavior model, which provides a topology representation of how the system is composed and operated by users. Despite its flexible features, this language is short of verification support. In this work, we apply model checking technique to MP language for the first time to support a wide range of properties checking including deadlock-freeness, reachability, and Linear Temporal Logic (LTL) properties.

In this paper, we present an automated approach to the modeling and verification of system architectures in the PAT framework [15], [22], [14]. Firstly, we cover a rich set of MP 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. Based on the formal semantics, we further developed a dedicated model checker based on PAT verification framework, which supports modeling, simulation and verification of MP models. Finally, we demonstrate our approach with the architecture modeling and verification of the client server, pipe filter and radar weapon system [6], where the effectiveness and useability of our approach are evaluated.

Related Work In the past decade, model checking techniques have been applied to software architecture designs [25], which aimed at achieving precise specification and rigorous verification of the intended structures and behaviors in the design. The advantage of verification is to determine whether a modeled structure can 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 [2], [3], ACME [9], and CHAM [8], [11]. Wright

and ACME capture the properties and structures of systems by composing components which interact through connectors, whereas, CHAM models system architecture in terms of molecules and transformation rules. The drawback of many existing approaches 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 the explicit use of parameterizing the specific behaviors of a particular type. This language is partially encoded into the FDR model checker [17], where only a subset of the language is supported and the verification is limited by the FDR tool (e.g., only compatibility checking and deadlock analysis are available). In comparison, much more properties such as reachability, and Linear Temporal Logic (LTL) properties checking are supported in our approach. Our recent work [24] on direct verification of extended Wright language improves [2], [3] with support of reachability and LTL. For ACME language, it is intended to support mapping from one architecture description language to an intermediate logical formalism and adopts an open semantic framework to reason about the model. Kim and Garlan [13] proposed the modeling and verification of architecture styles using the Alloy language and analyzer. In their approach, a few architecture styles based on ACME descriptions were translated to Alloy and verified. 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. In our approach, we support the simulation and verification of MP models directly without extra transitions which can save the overhead to a large extent compared with the translation approach adopted by Wright and ACME languages. The CHAM language has an effective way to express system properties but with no verification support. In addition to the above mentioned specific architecture description languages, a considerable amount of work has involved the Z [19] specification language and CSP [10] language. Z language is a model-based, set-theoretic formalism which is developed to be highly expressive. The system described in Z notations includes a set of system states and different operations. There are also the Object-Z [18] and TCOZ [16] languages proposed to extend Z with object-oriented styles and timing primitives. In contrast, CSP language is one of the process algebra languages. It defines system behaviors based on process modeling and formal reasoning about these models. The Wright language uses a subset of CSP to specify the behaviors of connected elements of a system. In particular, the concept of schemas in Z notations and the process expressions of CSP language provide inspiration and insight for formalizing and refining the behavior models in MP.

Organization The rest of the paper is organized as follows. Section II introduces the basic concepts and language features of MP language. Section III defines the syntax and operational semantics for MP. Section IV illustrates different properties we can verified based on MP models. Section V demonstrates several case studies using the MP model checker with evaluation results. Section VI concludes the paper and discusses the future work.

II. Basic Concepts of MP

In this section, we 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. One event should happen 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. Two events may happen concurrently if they are not ordered. The basic relations are transitive, non-commutative, non-reflexive, and distributive.

A. Event Grammar

The structure of event trace is specified by the event grammar rules in terms of *PRECEDES* and *IN* relations. The grammar rules have a form of:

A : *right-hand-part*;

where A stands for event type name. The following event patterns are used in the right hand part of grammar rules, where B, C, D stand for event type names or event patterns.

1) $A : B\ C\ D$;

The first event pattern is a sequence which represents the ordering of events under the *PRECEDES* relation. This rule means an event of a type A event contains ordered events b, c, and d, matching B, C, and D event patterns. Events b, c, d are *IN* event a, event b *PRECEDES* event c, and event c *PRECEDES* event d. The sequence pattern may contain any finite number of events, like "$A : B$; " or "$A : B\ C\ D\ E$; ".

Events are visualized by small squares, and the two basic relations are visualized by arrows. Specifically, the *PRECEDES* relation is denoted by solid arrow and the *IN* relation is denoted by dotted arrow. Figure 1 depicts the event trace specified by this rule:

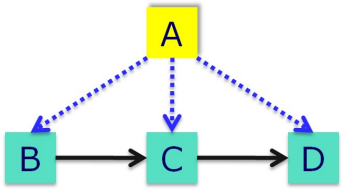

Fig. 1. An example of event trace defined by the Sequence Pattern

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* the 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 event trace for this rule is shown in Figure 2:

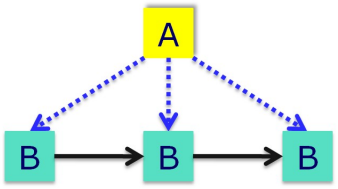

Fig. 2. An example of event trace defined by the Iterative Pattern

3) $A : \{B, \ C, \ D\}$;

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 of the types B, C and D, correspondingly. The events b, c, d are all *IN* event a. The set pattern may contain any number of event patterns, like "$A : \{B, \ C\}$; ", "$A : \{B, \ C, \ D, \ E\}$; ". The event trace in Figure 3 specifies a valid scenario for this rule:

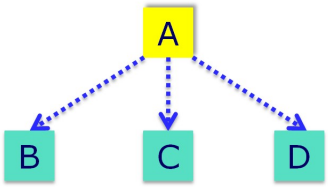

Fig. 3. An example of event trace defined by the Set Pattern

4) $A : \{* \ B \ *\}$;

The fourth rule denotes a set of zero or more events satisfying event pattern 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:

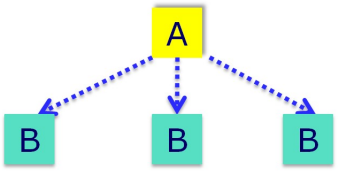

Fig. 4. An example of event trace defined by the Scope Set

5) $A : [\ B \ ]$;

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

Fig. 5. Optional Pattern

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

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

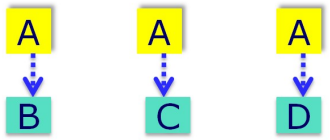

Fig. 6. Examples of event traces defined by the Alternative Pattern

The behavior model of a software system is specified using a set of event traces together with some constraints organized in a *schema*. The concepts of MP *schema* is inspired by Z schemas and the architectural concept of *configuration*. For a traditional *configuration*, it usually contains a collection of components and connectors, where components capture the behavior of each part of the system and connectors specify the interactions among components. In terms of MP model, both components and connectors are expressed by *root* events, while other events are used to specify the event structures and interactions. The detailed structure of MP *schema* will be introduced in Section III.

B. Share All

In addition to the basic grammar rules, MP also provides a mechanism for synchronizing root event behaviors through specific *share all* constraint. This operation plays a role similar to the event synchronization in CSP. The following shows two examples 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 first one means that $\{x : D \ | \ x \ IN \ A\} = \{y : D \ | \ y \ IN \ B\}$. The second constraint denotes that $\{x : D \ | \ x \ IN \ A \ or \ x \ IN \ B\} = \{y : D \ | \ y \ IN \ C\}$. We will use a *Multiple Synchronized Transactions* schema given in Figure 7 to illustrate this constraint. This MP schema requires that the *TaskA* and *TaskB* components are involved in a strictly synchronized communication. Each *Send* event

can only appear when the previous *Receive* event has been accomplished. A valid event trace specified by this example is shown in Figure 8 in the case for scope 2.

SCHEMA *Multiple Synchronized Transactions*
ROOT *TaskA* : $(*\ Send\ *)$;
ROOT *TaskB* : $(*\ Receive\ *)$;
ROOT *Connector* : $(*\ Send\ Receive\ *)$;
TaskA, *Connector* **SHARE ALL** *Send*;
TaskB, *Connector* **SHARE ALL** *Receive*;

Fig. 7. MP Codes for Multiple Synchronized Transactions schema

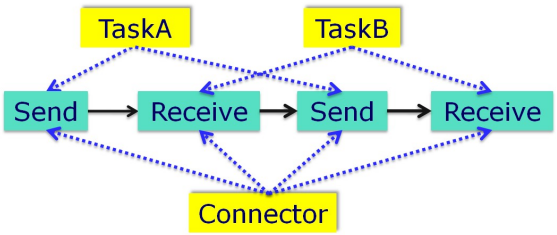

Fig. 8. Example of event trace for Multiple Synchronized Transactions schema

C. Slice

For the assertion language, MP proposes a useful clause *Slice* to represents a set of concurrent events with in the event trace.

$$Concurrent(x,y) \equiv \neg(x\ PRECEDES\ y) \wedge \neg(y\ PRECEDES\ x)$$

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

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

It can be viewed as a special relation used in assertions.

III. Formal Syntax and Semantics of MP

After introducing the basic concepts of MP language, we are ready to present the formal syntax and operational semantics in this section. As introduced in Section II, system behaviors in MP are described based on behavior models. Different event traces can be extracted from the well built behavior model. Each trace represents a valid execution case of the abstract system machine. Events in MP are the basic elements and defined at different levels of granularity. According to the language features, we classify all events into three categories. They are root events, middle events and leaf events respectively. Among them, root events and middle events can be viewed as composite events defined by pattern-lists; leaf events are the atomic events, which are executed at each step during the process of simulation or verification. Both middle events and leaf events can appear in the pattern-lists. The behavior model in MP is expressed via the *MP schema* which is organized as a set of root events without *PRECEDES* or *IN* relations between them. Different root events can be synchronized through the *share all* constraint.

A. Syntax of MP Schema

The syntax of MP schema is formalized in this subsection. We start with the formalization of the *MP Schema* definition as follows.

Definition 1 (MP Schema): A MP Schema is a 4-tuple $S = (Var, Init, P, C)$, where *Var* is a set of global variables; *Init* is the initial valuation of the variables; *P* denotes all the pattern-lists of root events; *C* is the *share all* constraint.

In the above *Schema*, multiple root events can have several leaf events in common. The common leaf events are provided by *share all* constraint. All the root events should execute their common leaf events simultaneously. For the respective other events, they are executed in interleave order. The structure of root event is defined via the *When* structure which has a form as below:

$$e_r ::= P\ when\ (e_1 \Rightarrow Q_1, \cdots, e_n \Rightarrow Q_n) \qquad \text{– when structure}$$

where e_r is the root event, e_i $(1 \leq i \leq n)$ is the leaf event, P and Q_i $(1 \leq i \leq n)$ denote pattern-lists. 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. Event e_i $(1 \leq i \leq n)$ may be inserted in the event trace at any place within the root trace, then the trace is continued with events specified by pattern-list Q_i. Users can also attach a *RESTART* clause after the *When* construct so as to resume the trace from the beginning of the pattern-list for the root after the interruption.

In MP, the behavioral aspects of the root events or middle events are described through pattern-list which is the key part of this language. Next, we will show the syntax of pattern-list. Most of the syntax is derived from the event grammar rules given in Section II.

$$\begin{array}{lll}
P ::= e_l & & \text{– atomic leaf event} \\
\quad | \ \{program\} & & \text{– special event} \\
\quad | \ Skip & & \text{– termination event} \\
\quad | \ P\ Q & & \text{– sequence} \\
\quad | \ (P_1 \mid P_2 \mid .... \mid P_n) & & \text{– alternative} \\
\quad | \ \{P_1, P_2, ..., P_n\} & & \text{– set} \\
\quad | \ \{* < a_1 - a_2 > P*\} & & \text{– scope set} \\
\quad | \ (* < a_1 - a_2 > P*) & & \text{– iterative} \\
\quad | \ if\ b\ \{P\}\ else\ \{Q\} & & \text{– conditional choice} \\
\quad | \ while\ b\ \{P\} & & \text{– while loop} \\
\quad | \ ref\{e_m\} & & \text{– middle event reference}
\end{array}$$

where e_l denotes the leaf event which is atomic and indivisible, e_m denotes middle event, b is a boolean expression, the capital letters P and Q represent pattern-lists. In addition, we need to define the scope for the scope set and iterative pattern-lists

in order to make the event trace finite when doing model checking. Therefore, a_1 and a_2 are two nonnegative integers that define the lower bound and upper bound of the scope.

An atomic leaf event e_l is executed individually. Special event $\{program\}$ is a paragraph of sequential program which is a statement block containing variable assignment, if-then-else or while structures. The right hand part of assignment is arithmetic or boolean operation. Special event is considered as an atomic event and executed in one step. The *Skip* event is a successful termination event. This event means a pattern-list finishes successfully without deadlock. Both special event and termination event are regraded as leaf events. $P\ Q$ is the sequence pattern. It behaves as pattern-list P first until its termination and then behaves as pattern-list Q. The alternative pattern $(P_1 \mid P_2 \mid .... \mid P_n)$ is made internally and non-deterministically where any P_i $(1 \leq i \leq n)$ may execute subsequently. The set pattern $\{P_1, P_2, ..., P_n\}$ denotes interleaving execution where any P_i $(1 \leq i \leq n)$ may perform their local actions without referring to each other. The scope set $\{* < a_1 - a_2 > P*\}$ defines several same pattern-list P execution without *PRECEDES* order. In contrast, iterative pattern $(* < a_1 - a_2 > P*)$ defines several pattern-list P execution in *PRECEDES* order, where each P must perform when the previous one has been finished. The number of pattern-list P must fall in the predefined scope in both iterative patterns. The *if* b $\{P\}$ *else* $\{Q\}$ is a conditional branching, when the boolean expression b is evaluated to be true, the system performs P, else performs Q. All the variables in expression b are global variables. Similarly, the while loop *while* b $\{P\}$ behaves continuously as pattern-list P or finishes the while loop immediately according to the value of b. A middle event can be referenced in the pattern-list.

We present two MP schemas to illustrate the syntax more clearly. One is the *Client_Server* schema in Figure 9 and the other is the *Pipe_Filter* schema in Figure 10. Both of the schemas are defined in the scope of case 2. In the *Client_Server* schema, the *Client* can request information from the *Server* and then block itself to wait for the reply. When the *Server* receives requests, it will process them and send back the results to *Client*. After receiving the results, the *Client* will unblock itself and continue executing. The *Connector* is used to restrict the order of event between *Client* and *Server*. This structure is quite commonly used in nowadays applications such as the Browser/Server structure. The second one is the *Pipe_Filter* schema which models a system whose execution is driven by data flow. There are two *Filter* components and one *Pipe* component in this system. Both of the *Filters* can receive data, process data and send out data. The *Pipe* is responsible for transmitting data from one *Filter* to another and keep the data flow direction. The *Connector* here is restricting the order of event, where the *Filter_One* sending out data must be performed before the *Pipe* getting data in and the *Filter_Two* receives data only after the data is sent out from the *Pipe*. This structure is also popular which can find its applications in many industrial examples, such as the data flow applications, Map-reduce model in cloud computing and Yahoo! Pipes.

B. Operational Semantics

In this subsection, 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 accordingly. We start with the definition of system configuration. It captures the global system state during system executions.

Definition 2 (Configuration): A system *configuration* is composed of two components (V, P), where V is the current valuation of all global variables, P is the current pattern-list expression.

The operational semantics for pattern-list is presented as firing rules associated with each pattern-list construct. Let ΣP denote a set of shared events of pattern-list P defined by *share all* constraint. Let $\xi\{P_1 + \cdots + P_n\}$ denote a set of events shared by a union of pattern-lists from P_1 to P_n defined by *share all* constraint. If a leaf event $e_l \in \xi\{P_1 + \cdots + P_n\}$, we can derive that $e_l \in \Sigma P_i$ $(1 \leq i \leq n)$. For simplicity, a function $upd(V, prog)$, to which given a sequential *program* and valuation 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. ϵ denotes an empty event which performs no action.

Figure 11 illustrates the firing rules. In *event* rule, the model behaves as an atomic leaf event e_l. An atomic event is performed in one step during the process of simulation or verification. The *program* rule is defined for the special event which is a paragraph of sequential program. The system will update the values of global variables according to the semantics of the program in this rule. The *alternative* rule is a multiple choice rule. The system can choose any pattern-list P_i $(1 \leq i \leq n)$ to execute subsequently. In *sequence* rules ($sequence_1$ and $sequence_2$), the system executes pattern-list P first. When P is found to be finished, the model will continue to execute pattern-list Q. The three *share* rules ($share_1$, $share_2$ and $share_3$) define the behavior of the root pattern-lists under the *share all* constraint. The shared leaf events of multiple pattern-list unions should be executed simultaneously. For other unshared events, they are performed without specific order. Assuming that we have two pattern-list unions: $\{P_1 + \cdots + P_n\}$ and $\{Q_1 + \cdots + Q_m\}$, which share all leaf event e_l. When the system executes rule $share_3$, one pattern-list P_i $(1 \leq i \leq n)$ from union $\{P_1 + \cdots + P_n\}$ and the other pattern-list Q_j $(1 \leq i \leq m)$ from union $\{Q_1 + \cdots + Q_m\}$ are chosen to execute the event e_l simultaneously. If event e_l is not a shared event, it can be executed without any specific order, which is defined by rule $share_1$ and rule $share_2$. The two *when* rules ($when_1$ and $when_2$) denote a root pattern-list which can be interrupted by multiple pattern-lists. When the system executes the *when* rules, the first event of pattern-list

SCHEMA *Client_Server*
ROOT *Client* : $\{* < 2-2 >$ *Request_Info Receive_Result Executing* $*\}$ *Skip*;
ROOT *Server* : $\{* < 2-2 >$ *Receive_Con Processing Provide_Result* $*\}$ *Skip*;
ROOT *Connector* : $(* < 2-2 >$ *Request_Info Receive_Con Provide_Result Receive_Result* $*)$ *Skip*;
Client, *Connector* **SHARE ALL** *Request_Info*, *Receive_Result*;
Server, *Connector* **SHARE ALL** *Receive_Con*, *Provide_Result*;

Fig. 9. A Client Server Schema

SCHEMA *Pipe_Filter*
ROOT *Filter_One* : $\{* < 2-2 >$ *Get_Data Processing_Data Send_Data* $*\}$ *Skip*;
ROOT *Filter_Two* : $\{* < 2-2 >$ *Receive_Data Processing_Data Dispatch_Data* $*\}$ *Skip*;
ROOT *Pipe* : $\{* < 2-2 >$ *Data_In Data_Out* $*\}$ *Skip*;
ROOT *Connector* : $\{\{* < 2-2 >$ (*Send_Data Data_In*) $*\}$ $\{* < 2-2 >$ (*Data_Out Receive_Data*) $*\}$ $\}$ *Skip*;
Filter_One, *Connector* **SHARE ALL** *Send_Data*;
Filter_Two, *Connector* **SHARE ALL** *Receive_Data*;
Pipe, *Connector* **SHARE ALL** *Data_In*, *Data_Out*;

Fig. 10. A Pipe Filter Schema

Q_i $(1 \leq i \leq n)$ may be inserted in the event trace at any place within the root trace, then the trace is continued with events specified by pattern-list Q_i. For *restart* rule, users can choose to put a *RESTART* clause after the interruption pattern-list Q_i $(1 \leq i \leq n)$. It defines that the event trace can restart from the beginning of the root pattern-list after the interruption. The *iterative* rules ($iterative_1$ and $iterative_2$) describe a scope sequence operation which implies the pattern-list P can happen sequentially for a number of times. Integer a is randomly chosen between a_1 and a_2 in order to specify the number of iteration times. If the value of a is chosen to be 0, the system has no behavior. Otherwise, it will behave as rule $iterative_1$. The *set* rule denotes a set of various pattern-lists execute concurrently without an ordering relation between them. The pattern-lists in set are executed without *PRECEDES* order. The *middle* rule denotes that you can place a pattern-list P behind a middle event M. This rule captures the behavior of the *IN* relation. The middle event can be regarded as a pattern-list reference. The *scope* rules ($scope_1$ and $scope_2$) denote a number of pattern-list P execute concurrently without *PRECEDES* order. Rules $scope_1$ and $scope_2$ are defined according to the value of a. The two *condition* ($condition_1$ and $condition_2$) rules define how to execute the conditional choice. If expression b is evaluated to be true, the pattern-list P is executed, otherwise pattern-list Q is executed. The two *while* ($while_1$ and $while_2$) rules define how the while loop works. If the value of expression b is true, the model behaves continuously as pattern-list P. Otherwise, it will finish the loop.

The MP *Schema* is translated into the Label Transition System (LTS) to perform simulation and verification. The definition of LTS is given as below.

Definition 3 (Label Transition System (LTS)): 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 a leaf event, S and S' are system configurations before and after the transition.

The labeled transition relationship conforms to the operational semantics presented in Figure 11. A finite execution of MP model is a finite sequence of alternating states/events $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1} \rangle$ where $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all $0 \leq i \leq n$. The event trace of MP Schema is extracted from the execution sequence by excluding all the states $\langle e_0, e_1, \cdots, e_n \rangle$. Each trace can be viewed as a valid execution of the abstract MP machine.

IV. VERIFICATION

The *Process Analysis Toolkit* (PAT) [15], [22], [14] is designed to apply state-of-the-art model checking techniques for system analysis. Our MP model checker is dedicated implemented based on PAT framework to support the MP model analysis and verification. It 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) and refinement checking [20].

There are two types of properties we are concerned with

$$\frac{}{(V, e_l) \xrightarrow{e_l} (V, \epsilon)}\ [\ event\ ] \qquad \frac{}{(V, \{prog\}) \rightarrow (upd(V, prog), \epsilon)}\ [\ program\ ] \qquad \frac{1 \leq i \leq n, (V, P_i) \xrightarrow{e_l} (V', P_i')}{(V, (P_1 \mid P_2 \mid .... \mid P_n)) \xrightarrow{e_l} (V', P_i')}\ [\ alternative\ ]$$

$$\frac{(V, P) \xrightarrow{e_l} (V', P')}{(V, P\ Q) \xrightarrow{e_l} (V', P'\ Q)}\ [\ sequence_1\ ] \qquad \frac{(V, Q) \xrightarrow{e_l} (V', Q'), P = \epsilon}{(V, P\ Q) \xrightarrow{e_l} (V', Q')}\ [\ sequence_2\ ]$$

$$\frac{(V, P_i) \xrightarrow{e_l} (V', P_i'),\ 1 \leq i \leq n,\ e_l \notin \Sigma P_i}{(V, < P_1, \cdots, P_i, \cdots, P_n,\ Q_1, \cdots, Q_m >) \xrightarrow{e_l} (V', < P_1, \cdots, P_i', \cdots, P_n,\ Q_1, \cdots, Q_m >)}\ [\ share_1\ ]$$

$$\frac{(V, Q_j) \xrightarrow{e_l} (V', Q_j'),\ 1 \leq j \leq m,\ e_l \notin \Sigma Q_j}{(V, < P_1, \cdots, P_n,\ Q_1, \cdots, Q_j, \cdots, Q_m >) \xrightarrow{e_l} (V', < P_1, \cdots, P_n,\ Q_1, \cdots, Q_j', \cdots, Q_m >)}\ [\ share_2\ ]$$

$$\frac{(V, P_i) \xrightarrow{e_l} (V', P_i'), (V, Q_j) \xrightarrow{e_l} (V', Q_j'),\ 1 \leq i \leq n,\ 1 \leq j \leq m, e_l \in \xi\{P_1 + \cdots + P_n\},\ e_l \in \xi\{Q_1 + \cdots + Q_m\}}{(V, < P_1, \cdots, P_i, \cdots, P_n,\ Q_1, \cdots, Q_j, \cdots, Q_m >) \xrightarrow{e_l} (V', < P_1, \cdots, P_i', \cdots, P_n,\ Q_1, \cdots, Q_j', \cdots, Q_m >)}\ [\ share_3\ ]$$

$$\frac{(V, P) \xrightarrow{e_l} (V', P')}{(V, P\ when\ (Q_1, \cdots, Q_n)) \xrightarrow{e_l} (V', P'\ when\ (Q_1, \cdots, Q_n))}\ [\ when_1\ ] \qquad \frac{(V, Q_i) \xrightarrow{e_l} (V', Q_i'),\ 1 \leq i \leq n}{(V, P\ when\ (Q_1, \cdots, Q_i, \cdots, Q_n)) \xrightarrow{e_l} (V', Q_i')}\ [\ when_2\ ]$$

$$\frac{(V, Q_i) \xrightarrow{e_l} (V', Q_i'),\ 1 \leq i \leq n}{(V, P\ when\ (Q_1, \cdots, Q_i[RESTART], \cdots, Q_n)) \xrightarrow{e_l} (V', Q_i'\ P\ when\ (Q_1, \cdots, Q_i[RESTART], \cdots, Q_n))}\ [\ restart\ ]$$

$$\frac{(V, P) \xrightarrow{e_l} (V', P'),\ 0 \leq a_1 \leq a \leq a_2,\ a \neq 0}{(V, (* < a_1 - a_2 > P *)) \xrightarrow{e_l} (V', P' \underbrace{P\ \cdots\ P}_{a-1})}\ [\ iterative_1\ ] \qquad \frac{0 \leq a_1 \leq a \leq a_2,\ a = 0}{(V, (* < a_1 - a_2 > P *)) \rightarrow (V, \epsilon)}\ [\ iterative_2\ ]$$

$$\frac{(V, P_i) \xrightarrow{e_l} (V', P_i'),\ 1 \leq i \leq n}{(V, \{P_1, \cdots, P_i, \cdots, P_n\}) \xrightarrow{e_l} (V', \{P_1, \cdots, P_i', \cdots, P_n\})}\ [\ set\ ] \qquad \frac{M \mathrel{\widehat{=}} P,\ (V, P) \xrightarrow{e_l} (V', P')}{(V, M) \xrightarrow{e_l} (V', P')}\ [\ middle\ ]$$

$$\frac{(V, P) \xrightarrow{e_l} (V', P'),\ 0 \leq a_1 \leq a \leq a_2,\ a \neq 0}{(V, \{* < a_1 - a_2 > P *\}) \xrightarrow{e_l} (V', \{P', \underbrace{P, \cdots, P}_{a-1}\})}\ [\ scope_1\ ] \qquad \frac{0 \leqslant a_1 \leq a \leq a_2,\ a = 0}{(V, \{* < a_1 - a_2 > P *\}) \rightarrow (V, \epsilon)}\ [\ scope_2\ ]$$

$$\frac{V \vDash b, (V, P) \xrightarrow{e_l} (V', P')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e_l} (V', P')}\ [\ condition_1\ ] \qquad \frac{V \nvDash b, (V, Q) \xrightarrow{e_l} (V', Q')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{e_l} (V', Q')}\ [\ condition_2\ ]$$

$$\frac{V \vDash b, (V, P) \xrightarrow{e_l} (V', P')}{(V, while\ b\ \{P\}) \xrightarrow{e_l} (V', P'\ \ while\ b\ \{P\})}\ [\ while_1\ ] \qquad \frac{V \nvDash b}{(V, while\ b\ \{P\}) \rightarrow (V, \epsilon)}\ [\ while_2\ ]$$

Fig. 11. Firing Rules

in model checking. One is the safety property which guarantees nothing bad happens. Examples of safety properties are deadlock-freeness and reachability checking. The other one is the liveness property which checks whether something good eventually happens. For this property, Linear Temporal Logic is a good candidate because the MP model makes explicit use of the events, states and variables. Moreover, the LTL provides a very intuitive and very mathematically precise notation for expressing properties about the Linear Temporal relation between the states/events in execution [7].

In MP model checker, we implement two searching strategies: Depth-first-search (DFS) and Breadth-first-search (BFS) to support the deadlock-freeness and reachability checking. A MP model is deadlock-free if and only if there does not exist a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that s_{n+1} is a deadlock state (i.e., a state at which no firing rules are applicable). Given a proposition p, a state satisfying the predicate is reachable (or equivalently p is reachable) if and only if there exists a finite execution $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that $s_{n+1} = (V_{n+1}, P_{n+1})$ and $V_{n+1} \vDash p$. Compared with the former two, the Linear Temporal Logic is relatively complex but is very useful in checking liveness properties. Given the *Client Server* structure, we want to verify whether each *Request_Info* event performed by *Client* will be responded with a *Provide_Result* event from *Server* eventually. This property could be stated as below, where $\Box$ and $\Diamond$ are modal operators which denote 'always' and 'eventually' respectively.

$$\Box\ (\mathit{Request\_Info}\ \Rightarrow \Diamond \mathit{Provide\_Result})$$

Such properties are very important in demonstrating the normal operations of systems. The integrated LTL formula [21] is defined as follows:.

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

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.

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

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 MP model checker, users can verify LTL properties with or without fairness. Moreover, different levels of fairness are supported including global fairness, event-level strong fairness and weak fairness. Interested readers could refer to [23] and [22] for more details.

V. Case Study and Evaluation

A. Modeling and Verifying Radar Weapon System

In this section, we apply our approach to model and verify the Radar Weapon system [6] to demonstrate the MP language as well as the MP model checker implemented in PAT. The Radar Weapon system is described via five components. Each of them represents a subsystem modeled by root event. The five subsystems are Generator, Radar, Weapon, Control and Enemy Missile respectively. The Generator is in charge of supplying power for Radar and Weapon when both of them are deployed. The Radar is responsible for detecting the Enemy missile. If the enemy is detected, it will activate the Weapon. When the Weapon attacks the enemy, it can either hit or miss it. The Control subsystem is used to coordinate the behaviors of Generator and Radar. The Radar must start working only after the Generator is launched. The environment is represented by Enemy Missile which may either approach or hit any of Generator, Radar, or Weapon. If the Generator is hit, the consequence is causing termination of energy production or consumption correspondingly. If the Radar and Weapon are hit, they can be repaired and resume work afterwards. The MP code of this system is given in Figure 12.

In this system, if the Generator is hit by the Enemy Missile, the deployed Radar, Weapon will get in the critical state of missing the power supply. If the system is in the critical state, the Radar and Weapon should stop working in a real word situation. Therefore, we want to guarantee that after the system entering critical state, the Radar and Weapon cannot be launched anymore. In order to check the designed system, we express the above properties in LTL formulaes as follows:

$$\Box(\mathit{Generator\_hit} \Rightarrow\ !(\Diamond \mathit{Weapon\_On}))$$
$$\Box(\mathit{Generator\_hit} \Rightarrow\ !(\Diamond \mathit{Radar\_On}))$$

The first property means whenever the Generator is hit, the Weapon will not be launched eventually. Similarly, the second property means whenever the Generator is hit, the Radar can not be launched eventually. We verify these two properties in MP model checker. Both of the verification are done in the scope of case 2. The verification results reveal that the first property is valid. However, the second property is not satisfied. One counterexample is displayed as follows:

$$\begin{array}{l}
\mathit{Approaching} \rightarrow \mathit{Approaching} \rightarrow \mathit{Generator_hit} \\
\rightarrow \mathit{Generator_Off} \rightarrow \mathit{Repair} \rightarrow \mathit{Idle} \rightarrow \mathit{Idle} \\
\rightarrow \mathit{Idle} \rightarrow \mathit{Idle} \rightarrow \mathit{Generator_On} \rightarrow \mathit{Radar_On}
\end{array}$$

From the trace of the counterexample, we can see that after being hit, the Generator can be repaired and restart work. Therefore the Radar can be launched after the Generator is hit. Whereas, the *Weapon_On* event is simultaneously synchronized by the Radar, Weapon and Enemy Missile. Even if the Radar is launched, the Weapon still cannot be triggered after the Generator is hit. We can modify the system schema by removing the *Repair* event and the *RESTART* clause from the Generator root event to make the properties of the system desirable. The modification means that the Generator cannot be repaired after it is hit by Enemy Missile. We continue to verify the above two properties in MP model checker, both of the properties turn out to be valid in this situation.

B. Performance Evaluation

We conducted experiments on the client server, pipe filter and radar weapon system to evaluate the performance. In order to compare with other tools, we model the client server and

SCHEMA *RadarWeaponSystem*
// ========== *Root*
ROOT *Generator* : $(*\ <2-2>$ *Idle Generator_On Generating Generator_Off* $*)$
WHEN { *Generator_hit* $\Rightarrow$ *Generator_Off Repair* [*RESTART*] };
ROOT *Radar* : $(*\ <2-2>$ *Idle Radar_On Radar_Working Radar_Off* $*)$
WHEN { *Radar_hit* $\Rightarrow$ *Radar_Off Repair* [*RESTART*] };
Radar_Working : $(*\ <2-2>$ (*Target_detected* | *No_target*) $*)$;
Target_detected : *Weapon_On*;
ROOT *Weapon* : $(*\ <2-2>$ (*Idle* | *Weapon_On Shoot Recharge*) $*)$
WHEN { *Weapon_hit* $\Rightarrow$ *Repair* [*RESTART*] };
Shoot : (*Hit* | *Miss*);
ROOT *Control* : $(*\ <2-2>$ *Generator_On Radar_On Monitoring*
Radar_off Generator_Off $*)$
WHEN { *Generator_hit* $\Rightarrow$ *Generator_Off Repair* [*RESTART*] ,
Radar_hit $\Rightarrow$ *Radar_Off Repair Radar_On* [*RESTART*] };
ROOT *Enemy_missile* : $(*\ <2-2>$ (*Approaching* | *Target_detected*) $*)$ *Boom*
WHEN { *Hit* $\Rightarrow$ *Win* };
Boom : (*Generator_hit* | *Radar_hit* | *Weapon_hit* | *Miss*);
// ========== *Constraint*
Radar, *Weapon*, *Enemy_missile* **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*
#**assert** *RadarWeaponSystem* $|= \Box($*Generator_hit* $\Rightarrow !(\Diamond$*Weapon_On*$))$;
#**assert** *RadarWeaponSystem* $|= \Box($*Generator_hit* $\Rightarrow !(\Diamond$*Radar_On*$))$;

Fig. 12. MP code of The Radar Weapon System

pipe filter structures both in PAT and Alloy [12]. In case of the client server structure, the property that each request performed by the client will be responded by the server eventually is verified. For the pipe filter structure, the property that data received by one filter is transferred to and dispatched by another filter is checked. Table I 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. The number of states and transitions are acquired in PAT. The experiments are done in scope of different cases. From the table, we can see PAT performs better than Alloy in most cases. This is because the operational semantics are directly implemented in PAT without extra transitions. Furthermore, PAT adopts explicit model checking which can handle 10^9 number of states in hours. Alloy uses SAT solvers as the verification engine, which is often less scalable for big systems and the performance is highly constrained by the capability of the SAT solvers. The models can be precisely constructed through MP schemas. But in Alloy, users have to explicitly model the relations between events which is complex and tedious.

We further evaluate the performance of our approach through much bigger cases of the radar weapon system. The experiment results is displayed in Table II, where one of the LTL properties that whenever the Generator is hit, the Weapon will not be launched eventually is verified in PAT. The results show that PAT performs reasonably well and can handle large scope cases in short time. Models used in the experiments can be download from the web page http://www.comp.nus.edu.sg/~pat/mp/. PAT can be downloaded from [1]. Note that we compare PAT with Alloy only because there is no other verification support for MP.

VI. Conclusions and Future Work

In this paper, we presented 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 modeling system and environment behaviors based on event traces, as well as supporting different architecture composition operations and views. Based on the formal semantics we implemented a dedicated model checker for MP in the PAT framework. Finally, we demonstrate the effectiveness of our approach through modeling and verification of client server, pipe filter and

Model	Scope	Property	Results	States/Transitions	PAT(Sec)	Alloy(Sec)
Client Server	3	$\Box(Request\_Info \rightarrow \Diamond Provide\_Result)$	valid	366/690	0.04	0.12
Client Server	4	$\Box(Request\_Info \rightarrow \Diamond Provide\_Result)$	valid	2751/5932	0.22	2.07
Client Server	5	$\Box(Request\_Info \rightarrow \Diamond Provide\_Result)$	valid	21522/53160	1.44	-
Pipe Filter	2	$\Box(Get\_Data \rightarrow \Diamond Dispatch\_Data)$	valid	426/1041	0.11	0.09
Pipe Filter	3	$\Box(Get\_Data \rightarrow \Diamond Dispatch\_Data)$	valid	3132/10180	0.36	2.60
Pipe Filter	4	$\Box(Get\_Data \rightarrow \Diamond Dispatch\_Data)$	valid	34702/159655	2.94	-

TABLE I
EXPERIMENT RESULTS OF COMPARING PAT WITH ALLOY

Model	Scope	Property	Results	States	Transitions	Time(Sec)
Radar Weapon	6	$\Box(Generator\_hit \rightarrow\ !(\Diamond Weapon\_On))$	valid	355466	1435108	47.24
Radar Weapon	7	$\Box(Generator\_hit \rightarrow\ !(\Diamond Weapon\_On))$	valid	619127	2521028	94.58
Radar Weapon	8	$\Box(Generator\_hit \rightarrow\ !(\Diamond Weapon\_On))$	valid	1006569	4125666	178.70
Radar Weapon	9	$\Box(Generator\_hit \rightarrow\ !(\Diamond Weapon\_On))$	valid	1551272	6391642	255.28

TABLE II
EXPERIMENT RESULTS OF HANDLING LARGE SCOPE CASES IN PAT

radar weapon system. In addition, performance evaluations were presented to measure the scalability of the approach.

In the future, we plan to extend 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 MP model checker. The GUI should provide diagram representations of the architecture models as well as support the definitions of the formal specifications. In addition, we can further our work via designing an architecture style library 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.

REFERENCES

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

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

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

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

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

[6] M. Auguston and C. Whitcomb. System architecture specification based on behavior models. In *Proceedings of the 15th ICCRTS Conference (International Command and Control Research and Technology Symposium)*, Santa Monica, CA, June 22-24 2010.

[7] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

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

[9] D. Garlan, R. T. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON*, page 7, 1997.

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

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

[12] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[13] J. S. Kim and D. 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] Y. Liu, J. Sun, and J. S. Dong. An Analyzer for Extended Compositional Process Algebras. In *ICSE Companion*, pages 919–920. ACM, 2008.

[15] Y. Liu, J. Sun, and J. S. Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199, 2011.

[16] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 95–104, 1998.

[17] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[18] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

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

[20] J. Sun, Y. Liu, and J. S. Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.

[21] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *TASE*, pages 127–135, 2009.

[22] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *Proceedings of the 21th International Conference on Computer Aided Verification (CAV 2009)*, pages 702–708, June 2009.

[23] 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.

[24] J. X. Zhang, Y. Liu, J. Sun, J. S. Dong, and J. Sun. Model Checking Software Architecture Design. In *HASE*, June 2012. accepted.

[25] P. Zhang, H. Muccini, and B. Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5):723–744, 2010.