Model Checking Software Architecture Design

Jiexin Zhang*, Yang Liu†, Jing Sun‡, Jin Song Dong* and Jun Sun§
*Department of Computer Science, School of Computing, National University of Singapore, Singapore
Emails: {jiexinzh,dongjs}@comp.nus.edu.sg
†School of Computer Engineering, Nanyang Technological University, Singapore
Email: yangliu@ntu.edu.sg
‡Department of Computer Science, The University of Auckland, New Zealand
Email: j.sun@cs.auckland.ac.nz
§ISTD, Singapore University of Technology and Design, Singapore
Email: sunjun@sutd.edu.sg

***Abstract*—Software Architecture plays an essential role in the high level description of a system design. 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 paper, we present an automated approach to the modeling and verification of software architecture designs using the Process Analysis Toolkit (PAT). We present the formal syntax of the Wright# architecture description language together with its operational semantics in Labeled Transition System (LTS). A dedicated model checking module for Wright# is implemented in the PAT verification framework based on the proposed formalism. The module – ADL supports verification and simulation of software architecture models in PAT. We advance our work via defining an architecture style library that embodies commonly used architecture patterns to facilitate the modeling process. Finally, a case study of the Teleservices and Remote Medical Care System (TRMCS) modeling and verification is presented to evaluate the effectiveness and scalability of our approach.**

***Index Terms*—Software Architecture; Formal Verification; Model Checking; Wright; PAT.**

I. Introduction

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. Despite its importance, the lack of formal description and verification support hinders the development of quality architectural models. The current practice of software architecture modeling mainly relies on diagrammatic notations and informal textual descriptions. In the past decade, formal modeling techniques have been applied to software architecture designs [2], 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 [2], [1], Darwin [11], ACME [4], CHAM [6] and MP [3]. Wright, Darwin and ACME capture the properties and structures of systems by introducing composed components interacted through connectors, where CHAM models system architecture in terms of molecules and transformation rules. MP proposes to formalize the architecture specification based on behavior models and event traces.

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 [10] to perform verification. The language can handle behavioral reconfiguration, well but cannot address the issues of complex interactions among reconfiguration unites. In the case of ACME language [4], 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 [7] 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. [18] 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 no verification

support. The MP language formalizes the system architecture via event traces as well as the event grammar rules. The behavior of systems is captured by a variety of events and also the precedence/inclusion relations. This language can hardly support architecture reconfiguration and reuse.

In this paper, we present an automated approach to the modeling and verification of software architecture designs in the PAT framework [9], [14], [8]. We proposed a new software architecture description language – Wright#, by combing several previous works and adding new features. The structural description of the language is extended from the Wright notation [2], [1], which was proposed by Allen and Garlan in 1997. The language is relatively mature, complete, and elegant compared with other architecture description languages. We adopt the *Configuration* structure of the language to capture the behaviors and interactions of each part of systems. The structure is clear and precisely designed to give users a topology representation of how the system is composed and operated. We extend Wright with a rich set of syntax to describe concurrent communications between the components and connectors. We formally define the syntax and operational semantics of Wright# to provide the foundation of formal analysis. The new language is capable of describing both static and dynamic system behaviors, as well as supporting the architecture style configuration and reuse. Based on the formal semantics, we further developed a dedicated model checking module for Wright# in the PAT verification framework, which supports modeling, simulation and verification of software architecture models.

Large and complex software systems are often represented using a combination of different architectural patterns (styles) [18]. In light of this observation, we built an architecture style library in the tool to facilitate the reuse of basic and common architecture patterns with extension. The library contains a set of commonly used software architecture styles, such as the client-server, peer-to-peer, pipe-filter, publish-subscriber, shared-data, etc. With the help of the style library, users are able to extend and reuse existing structures in their software architecture designs. Finally, we demonstrate our approach with a real world case study of a Teleservices and Remote Medical Care System (TRMCS) [5] architecture modeling and verification, where the effectiveness and scalability of the approach are evaluated.

The rest of the paper is organized as follows. Section II defines the syntax and operational semantics of Wright#. Section III presents the definition and verification of an architecture style library in PAT. Section IV demonstrates the modeling and verification of a real world case study using the architecture module in the PAT model checker with evaluation results. Section V concludes the paper and discusses the future work.

II. Formal Syntax and Semantics of Wright#

In this section, we present the formal syntax and operational semantics of the Wright# architecture description language. Our notation is heavily influenced by the Wright language, which adopts an architecture view of Component-and-Connector (C&C) [2].

A. Formal Syntax of Wright#

The syntax of Wright# is formalized in this subsection. We start with the formalization of the *Component* definition as follows.

Definition Component. A component is a 4-tuple $C = (Var_C, init_C, Computation, Ports)$, where Var_C is a set of variables; $init_C$ is the initial valuation of the variables; *Computation* defines the behavior of the component; *Ports* is a set of ports.

A component has two elements to express itself: one is a number of *Ports*, the other is the *Computation*. *Ports* serve as the interfaces for the component to communicate with its environment. The specification of each port indicates what interaction the component has involved in. A more comprehensive description of the properties of each *Component* is given by computation. Similarly, connectors have a *Glue* and a set of *Roles* as defined below to specify the links between components. The role explains how each participant joins the interaction. The *Glue* provides a complete description about how the participants work together to establish the connection. We can see that the function of *Glue* in a connector is similar to that of the *Computation* in a component. Variables inside the components or connectors are local, which cannot be accessed by other components or connectors.

Definition Connector. A connector is a 4-tuple $N = (Var_N, init_N, Glue, Roles)$, where Var_N is a set of variables; $init_N$ is the initial valuation of the variables; *Glue* defines the behavior of the connector; *Roles* is a set of roles.

Definition Configuration. A configuration model is a 5-tuple $\mathcal{C} = (Ch, C, N, I, A)$ where *Ch* is a set of channels; *C* is a set of components; *N* is a set of connectors; *I* is a set of instances of components and connectors; *A* defines the mapping from components' ports to connectors' roles.

In order to give a complete specification of the system architecture, we introduce a *Configuration* schema. Besides the components and connectors, configuration also includes the instance definitions *I* and the attachment definitions *A*. In the instances, users need to define a specific number of instances for each participated component and connector in the system. We can understand instances as the created objects of each predefined class in Object-Oriented programming languages like JAVA and C#. Each instance can inherent all the properties and computations of the component or connector. The components and connectors are more like types that can be reused in many actual examples. The name of each instance is required to be explicitly and uniquely defined to avoid conflicts. Following instances, there is only one part left for completing the configuration which is the attachment relations. In attachments, the components' ports are associated with the connectors' roles to form the whole system. Especially,

the attached ports and roles are required to be compatible in function. The components are only permitted to attach to connectors. Therefore, the components, connectors, instances and attachments altogether define the configuration of a whole system architecture.

The syntax to define the behaviors of components and connectors in Wright are based on a subset of CSP language. We propose a more complete syntax to model components and connectors. The constructs mainly include process, events, internal/external choices, and parallel composition. Compared with Wright, our extended syntax not only includes various concurrent communications, but also hierarchical control flows and a rich set of data structures and operations.

$$
\begin{array}{ll}
P = \mathit{Stop} \mid \mathit{Skip} & \text{– primitives} \\
\mid e\{\mathit{prog}\} \rightarrow P & \text{– event prefixing} \\
\mid \mathit{ch}!\mathit{exp} \rightarrow P \mid \mathit{ch}?x \rightarrow P & \text{– channel communication} \\
\mid P \ ; \ Q & \text{– sequential composition} \\
\mid P \ \Box \ Q & \text{– choice operator} \\
\mid \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\} & \text{– conditional choice} \\
\mid [b]P & \text{– state guard} \\
\mid P \parallel Q & \text{– parallel composition} \\
\mid P \mid\mid\mid Q & \text{– interleave composition} \\
\mid P \triangle Q & \text{– interrupt} \\
\mid \mathit{ref}(Q) & \text{– process reference}
\end{array}
$$

In the above, we present the syntax of architecture processes for describing the behavioral aspects of the *Ports*, *Computation*, *Roles* and *Glue*. Architecture processes are abbreviated as processes for simplicity in the sequel. P and Q are architecture processes, e is a simple event, ch is a channel, b is a boolean expression. In addition, $e!$ represents an event which sends out data to its environment, $e?$ represents an event which receives data from the environment. The process *Stop* represents the system entering a deadlock state, while *Skip* is a process that represents successful termination. Event prefixing $e \rightarrow P$ engages in event e first and then behaves as process P. If event e is attached with a program, the program will be executed together with the occurrence of event e. $P \ ; \ Q$, behaves as P until its termination and then behaves as Q. Choice $P \ \Box \ Q$ is resolved only by the occurrence of an event. Both $\mathit{if}\ b\ \{P\}\mathit{else}\{Q\}$ and $[b]P$ are conditional branchings. For the former, when b is evaluated to be true, the system performs P, else performs Q. For the latter, process P can be executed until b is evaluated to be true. Parallel composition of two processes with barrier synchronization is written as $P \parallel Q$, where P and Q may perform lock-step synchronization, i.e., P and Q execute an event simultaneously. Two processes which run concurrently without barrier synchronization is written as $P \mid\mid\mid Q$, where $\mid\mid\mid$ denotes interleaving. Both P and Q may perform their local actions without referring to each other. Process P interrupt process Q behaves as process P first until the first visible event of process Q is engaged, and then the control is transferred to Q. Given a channel ch with pre-defined buffer size, process $\mathit{ch}!\mathit{exp} \rightarrow P$ evaluates the expression exp (with the current valuation of the variables) and puts the value into the tail of the respective buffer and behaves as P. Process $\mathit{ch}?x \rightarrow P$ gets the top element in the respective buffer, assigns it to variable x and then behaves as P. Sending/receiving multiple messages at once is supported. If a channel has buffer size 0, it is a synchronous channel, whose input and output communications must occur synchronously. A process expression could be given a name for referencing to support the recursion in system model.

B. Operational Semantics of Wright#

In this section, we present operational semantics of the Wright# modeling language, 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 of a system model, we first define the notion of system state to capture the global system information during the executions. Due to the page limit, we only consider synchronous channel communication in this subsection, which removes the (bounded) channel buffer from the system state below and corresponding firing rules in Figure 1. But in our implementation, both synchronous and asynchronous channel communication are supported.

System state A system state is composed of two components (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 architecture process.

First of all, we present the operational semantics for architecture processes as firing rules associated with each process construct. Let Σ denote a set of events. For simplicity, a function $\mathit{upd}(V, \mathit{prog})$, 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. We write $\mathit{eva}(V, \mathit{exp})$ to denote the value of the expression evaluated with variable valuations in V.

Figure 1 illustrates the firing rules. Rule *prefix* captures how event associated with sequential programs are handled, i.e., the occurrence of the event and program is simultaneous and appears, to the system, to be atomic. Notice that, this is the only way global variables are modified. Rule *channel* captures the semantics of synchronous channel communication. We remark that there are two rules (con_1 and con_2) associated with $\mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\}$, whereas only one rule (*guard*) is associated with $[b]P$. Therefore, if b is false given $[b]P$, then the process is blocked until b becomes true. The semantics of parallel composition $P \parallel Q$ are captured using three rules. Either P or Q can make a move if the event x is not in their common alphabets (see rule p_1 and p_2), otherwise P and Q have to synchronize on x (see rule p_3). Function αP, also called the alphabet of P, returns the set of events that appear in process P. Notice that the event in a data operation is called *non-communicating event*, which is excluded from the alphabet in parallel processes.

$$\frac{V \vDash b, (V,P) \xrightarrow{e} (V',P')}{(V,[b]P) \xrightarrow{e} (V',P')}\ [\ guard\ ]$$

$$\frac{}{(V, e\{prog\} \rightarrow P) \xrightarrow{e} (upd(V,prog), P)}\ [\ prefix\ ]$$

$$\frac{(V,P) \xrightarrow{ch!} (V,P'), (V,Q) \xrightarrow{ch?} (V,Q')}{(V,P \parallel Q) \xrightarrow{ch} (V,P' \parallel Q')}\ [\ channel\ ]$$

$$\frac{P \mathrel{\widehat{=}} Q, (V,Q) \xrightarrow{x} (V',Q')}{(V,P) \xrightarrow{x} (V',Q')}\ [\ def\ ] \qquad \frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Stop)}\ [\ skip\ ]$$

$$\frac{V \vDash b, (V,P) \xrightarrow{x} (V',P')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{x} (V',P')}\ [\ con_1\ ]$$

$$\frac{V \nvDash b, (V,Q) \xrightarrow{x} (V',Q')}{(V, if\ b\ \{P\}\ else\ \{Q\}) \xrightarrow{x} (V',Q')}\ [\ con_2\ ]$$

$$\frac{(V,P) \xrightarrow{e\tau} (V',P')}{(V,P;\ Q) \xrightarrow{e\tau} (V',P';\ Q)}\ [\ seq_1\ ] \qquad \frac{(V,P) \xrightarrow{\checkmark} (V',P')}{(V,P;\ Q) \xrightarrow{\tau} (V',Q)}\ [\ seq_2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P')}{(V,P \Box Q) \xrightarrow{x} (V',P')}\ [\ ch_1\ ] \qquad \frac{(V,Q) \xrightarrow{x} (V',Q')}{(V,P \Box Q) \xrightarrow{x} (V',Q')}\ [\ ch_2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V',P'), x \in \alpha P, x \notin \alpha Q}{(V,P \parallel Q) \xrightarrow{x} (V',P' \parallel Q)}\ [\ p_1\ ]$$

$$\frac{(V,Q) \xrightarrow{x} (V',Q'), x \in \alpha Q, x \notin \alpha P}{(V,P \parallel Q) \xrightarrow{x} (V',P \parallel Q')}\ [\ p_2\ ]$$

$$\frac{(V,P) \xrightarrow{x} (V,P'), (V,Q) \xrightarrow{x} (V,Q'), x \in \alpha P \cap \alpha Q}{(V,P \parallel Q) \xrightarrow{x} (V,P' \parallel Q')}\ [\ p_3\ ]$$

where $e \in \Sigma$ and $x \in \Sigma \cup \{\checkmark\}$

Fig. 1. Wright# firing rules

To define the behavior of configuration schemas, we firstly define the behavior of components and connectors. A prefixing function $prefix(pre, P)$ is defined as adding prefix "pre :" to all the events and channel communications of process P. One example is shown below.

$$prefix(a, invoke? \rightarrow return! \rightarrow Provider) = a : invoke? \rightarrow a : return! \rightarrow prefix(a, Provider)$$

For a process definition $P \mathrel{\widehat{=}} Q$, we define $dprefix(P)$ as $prefix(name(P), Q)$, where $name(P)$ represents symbol "P :". One example is shown below.

$$dprefix(Consumer \mathrel{\widehat{=}} req! \rightarrow result? \rightarrow Consumer) = Customer : req! \rightarrow Customer : result? \rightarrow prefix(Customer, Customer)$$

Given a component $C = (Var_C, init_C, Computation, Ports)$, its behavior $\mathcal{P}_C$ is defined as a parallel composition of *Computation* and *Ports* with the application of *dprefix* function on each port. Similarly, for a connector $N = (Var_N, init_N, Glue, Roles)$, its behavior $\mathcal{P}_N$ is defined as a parallel composition of *Glue* and *Roles* with the application of *dprefix* on each role. The reason of adding the port name (or role name) as the prefix in the events in port process (or role process) is to synchronize with the events in *Computation* (or *Glue*) process. One example of a filter component in a Pipe-filter structure is shown below.

$$\mathcal{P}_{filter} = Glue \parallel dprefix(input) \parallel dprefix(output)$$

Given a configuration, for an instance declaration i : *component* (or i : *connector*), the behavior of i is defined as $\mathcal{I}_i = prefix(i, \mathcal{P}_{component})$ (or $\mathcal{I}_i = prefix(i, \mathcal{P}_{connector})$). For a mapping *a.p as b.r* from a component a port p to a connector b role r, the meaning for such mapping is to rename all the events prefixed with "$b : r$:" in $\mathcal{I}_b$ by replacing the prefix "$b : r$:" to "$a : p$:". The semantics of a configuration $\mathcal{C} = (Ch, C, N, I, A)$ is defined using a LTS as follows.

Configuration LTS Given a configuration $\mathcal{C}$ = *(Ch, C, N, I, A)*, let Σ denote the set of all events in all instances in I by applying the mapping rules in A. The labeled transition system corresponding to $\mathcal{C}$ is a 3-tuple $\mathcal{L}^{\mathcal{C}} = (S, init, \rightarrow)$, where S is a set of states, $init \in S_{\mathcal{C}}$ is the initial state $(init_{\mathcal{C}}, P_{\mathcal{C}})$ ($init_{\mathcal{C}}$ is the combination of all the valuations of all instances in I, $P_{\mathcal{C}}$ is a parallel composition of behaviors of all instances in I by applying the mapping rules in A), and $\rightarrow \subseteq S \times \Sigma \times S$ is a labeled transition relation.

Given an LTS $(S, init, \rightarrow)$, the size of S can be infinite for two reasons. First, the variables may have infinite domains or the channels may have infinite buffer size. We require (syntactically) that the sizes of the domains and buffers are bounded. Second, processes may allow unbounded replication by recursion, e.g., $P = (a \rightarrow P;\ c \rightarrow Skip) \Box b \rightarrow Skip$, or $P = a \rightarrow P \ ||| \ P$. In this work, we consider only LTSs with a finite number of states for the reason of applying model checking. In particular, we bound the sizes of value domains and the number of processes by constants. In our examples, bounding the sizes of value domains also bounds the depths of recursions.

As PAT is a generic verification framework, a dedicated model checking module – ADL, is implemented based on the above formal syntax and operational semantics of the Wright# language. The ADL module supports Wright# with featured model editor, animated simulator and verifier. The user friendly simulator can interactively and visually simulate 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 property analysis such as deadlock-freeness, divergence-freeness, reachability checking, Linear Temporal Logic (LTL) properties with or without fairness assumptions and refinement checking [12]. In the next section, we present a variety of architecture structure patterns that can be modeled by the ADL module.

III. Formalizing Architecture Style Library

In this section, we present the style library that we defined for software architecture module in PAT. The purpose of such a library is for the extension as well as reuse of the common structure patterns in a complex design. The style library we have built includes a set of basic architecture styles, e.g., Client-server, Peer-to-peer, Pipe-filter, Publish-subscriber, Shared-data, etc. In the following, we will introduce the styles one by one and give instructions on how to inherit them in a customized user model.

```
Style  Client_Server
{
   Component Client
   {
      Port Request = requestInfo!->receiveResults?->Request;
      Computation = Request:requestInfo!->Request:receiveResults?
                  ->Computation;
   }
   Component Server(i)
   {
      Port[i] Provide = ReceiveCon?->provideResults!->Provide;
      Computation = Provide[0]:ReceiveCon?
                  ->Provide[0]:provideResults!->Computation;
   }
   Connector Request_Reply_Con
   {
      Role Consumer = requestInfo!->receiveResults?->Consumer;
      Role Provider = ReceiveCon?->provideResults!->Provider;
      Glue = Consumer:requestInfo?->Provider:ReceiveCon!
           ->Provider:provideResults?->Consumer:receiveResults!->Glue;
   }
}
```

Fig. 2. A Client-server Style

Client-server style is a basic architecture style where there are two components: client and server, also with a *request/reply* connector communicating them. The style modeled in Wright# is shown in Figure 2. The client usually requests services or information from the server and then waits for the reply. During this period, the client does nothing but block itself. When the server receives request, it will process and send back the results to client. After receiving the results, client will unblock itself and continue executing. The connector in this style takes charge of transmitting request message from client to server and the returned results from server to client. Each client has one *request* port. Each server has an indefinite number of *provide* ports which permit an arbitrary number of clients connecting with it. Therefore, we add one parameter i to define the number of *provide* ports. When users extend this style, they should specify the concrete value of i. In the *request/reply* connector we set two roles to establish the connection – one is the *consumer* role, the other is the *provider* role. This connector has the function of restricting the data flow direction among components. Client-server style is commonly used in developing network related applications, such as the Browser/Server structure.

```
Style Pipe_Filter
{
  Component Filter(i, j)
  {
     Port[i] input = getData?->input;
     Port[j] output = putData!->output;
     Computation = input[0]:getData?->output[0]:putData!->Computation;
  }
  Connector Pipe
  {
     Role data_in = putData!->data_in;
     Role data_out = getData?->data_out;
     Glue = data_in:putData?->data_out:getData!->Glue;
  }
}
```

Fig. 3. A Pipe-filter Style

Pipe-filter style presents a system whose execution is driven by data flow. The style modeled in Wright# is shown in Figure 3. The components in this style are named filters. One filter can have multiple *input* ports to receive data and multiple *output* ports to send out data. The connector pipe is responsible for transmitting data from one filter's *output* port to another filter's *input* port. One pipe has a single *data_in* role and a single *data_out* role to connect two filters. The computation of pipe always preserves the data flow's sequence. In this style, the attachment relation associates the *output* ports of filters with the *data_in* roles of pipes and the *input* ports of filters with *data_out* roles of pipes. The number of *input* ports and *output* ports are set as two parameters. This makes the filters more flexible to adapt to user's model. This style can find its applications in many industrial examples, such as data flow applications, Map-reduce model in cloud computing and Yahoo! Pipes.

```
Style  Publisher_Subscriber
{
      Component Publisher
      {
            Port Generate = pub!->Generate;
            Computation = Generate:pub!->Computation;
      }
      Component Subscriber
      {
            Port Accept = deliver?->Accept;
            Computation = Accept:deliver?->Computation;
      }
      Connector Event_Con(m)
      {
            Role publisher = pub!->publisher;
            Role[m] subscriber = deliver?->subscriber;
            Glue = publisher:pub?->Transfer(); Glue;
            Transfer() = |||x:{0..(m-1)}@ Deliver(x);
            Deliver(i) = subscriber[i]:deliver!->Skip;
      }
}
```

Fig. 4. A Publish-subscriber Style

Publish-subscriber style is mainly applied to describe asynchronism systems. The style modeled in Wright# is shown in Figure 4. It contains two categories of components: publishers and subscribers. In this style, the publisher could publish events which are distributed to its subscribers. The connector takes charge of doing the distribution. Each publisher has one

generate port to generate events. Each subscriber has one *accept* port to receive a variety of events. The connectors in this style have *publisher* roles and *subscriber* roles to connect with *generate* ports and *accept* ports in components. In realistic applications, a publisher usually publishes events to multiple subscribers. Taking account of this situation, the number of subscribers is defined as a parameter in the connector. when users inherit this style, the value of parameter need to be set in accordance with the number of subscriber instances. Representative examples of this style include mailing systems and social networks, such as Facebook and Twitter.

```
Style Peer2Peer
{
   Component Peer
   {
      Port Client = requestInfo!->receiveResults?->Client;
      Port Server = ReceiveCon?->provideResults!->Server;
      Computation = client()|||server();
      client() = Client:requestInfo!->Client:receiveResults?->client();
      server() = Server:ReceiveCon?->Server:provideResults!->server();
   }
   Connector P2P_Con
   {
      Role clientA = requestInfo!->receiveResults?->clientA;
      Role clientB = requestInfo!->receiveResults?->clientB;
      Role serverA = ReceiveCon?->provideResults!->serverA;
      Role serverB = ReceiveCon?->provideResults!->serverB;
      Glue = clientA_serverB()|||clientB_serverA();
         clientA_serverB() = clientA:requestInfo?->serverB:ReceiveCon!->
    serverB:provideResults?->clientA:receiveResults!->clientA_serverB();
         clientB_serverA() = clientB:requestInfo?->serverA:ReceiveCon!->
    serverA:provideResults?->clientB:receiveResults!->clientB_serverA();
    }
}
```

Fig. 5. A Peer-to-peer Style

Peer-to-peer style is quite similar to Client-server style but is more complex. It distinguishes itself by allowing each peer to act as both server and client which means it can provide service and invoke service simultaneously. Therefore, each peer has one *client* port and one *server* port behaving actions just as in Client-server style. The style modeled in Wright# is shown in Figure 5. In the computation portion, the client process is executed interleaved with the server process. There are four roles defined in the connector, namely, *clientA*, *clientB*, *serverA*, and *serverB*. The relations among them are specified in the glue where *clientA* could request services from *serverB* and *clientB* could request services from *serverA*. Peer-to-peer style is becoming more and more popular in nowadays systems. Many file sharing and instant messaging systems like BitTorrent, eDonkey, MSN and Skype are all utilizing this architecture.

Other architecture styles such as Shared-Data, etc., can be modeled in a similar manner. All the styles are verified to be deadlock free and satisfy desirable safety and liveness properties, which are specified in the form of state/event LTL formulae. In the next section, we will demonstrate the use of the style library in modeling a complex system architecture.

IV. Case Study and Evaluation

In this section, we apply our approach to the modeling and verification of the Teleservices and Remote Medical Care System (TRMCS) [5]. The TRMCS system aims at providing

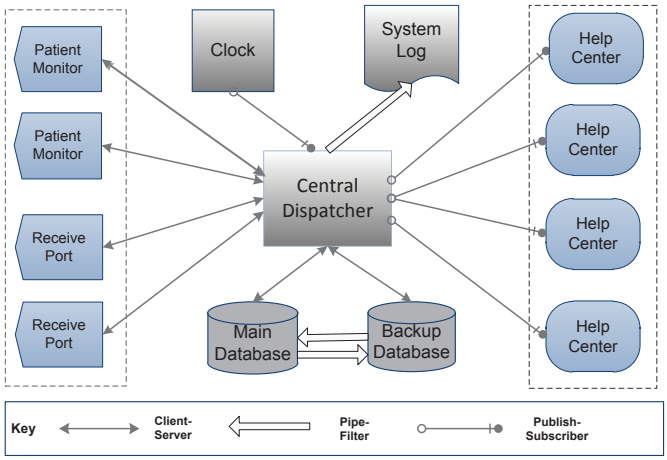

Fig. 6. The Component and Connector view of TRMCS

medical services to at-home users through the Internet or mobile phones. Synthetically, the system should provide the following capabilities:

1) Allow the user or monitoring software to issue help requests to the assistance center.
2) Guarantee the continuous service of the system.
3) Guarantee the delivery of help service in response to a help request in a specific critical time.
4) Handle several help requests in parallel that competes for service by overlapping in time and space.
5) Handle dynamic changes to the number and location of users.
6) Provide persistent repository of data and history log.

The component and connector view diagram in Figure 6 illustrates the overall architecture of TRMCS system. From the diagram, we can observe that the TRMCS consists of a multi-styled structure in its design, which includes client-server, pipe-filter and publish-subscriber. We will discuss details on the modeling and verification in the following section.

A. Modeling and Verifying TRMCS

The system could receive requests in two different ways according to the requirements. In one case, a patient with a medical emergency sends a help request to TRMCS system. The patient expects to receive a reply in certain critical time. In the other case, a patient may have internet-based medical monitors that give continuous readouts, e.g., EKG and EEG. A help center may be contracted to read these monitors over the net and raise alerts when dangerous values are detected. For the above two cases, we separately use the components Patient Monitors and Receive Ports to model their behaviors. When monitors detect dangerous values or ports receive user calls, they both can report requests to Central Dispatcher for further processing. The relationship between them and the dispatcher are client-server structures. The critical part of this system is the Central Dispatcher. It is in charge of organizing the other parts and responsible for communicating.

It can receive requests from all ports and monitors concurrently and process them in parallel. In order to check the validity of each request, the dispatcher need retrieve patient information from the patient database. Hence, the relationship between dispatcher and databases can be viewed as client-server structures. There are two types of patient database: Main Database and Backup Database. When the main database fails to response, the access should automatically switch to the backup database. The data inside the main and backup databases should always be synchronized to avoid inconsistency. The connections between two types of databases are pipe-filter structures. After receiving the requests, dispatcher will distribute them to help centers for proving emergency assistance. A request can be either picked by a help center actively or assigned by the dispatcher to help centers passively. Therefore, the help centers have to register in dispatcher and receive the published incident information from dispatcher. We can get that the relationship between the dispatcher and the help centers is a publisher-subscriber structure. In terms of each request, the system needs to record the request launch time and the time of assignment or selection. That means the dispatcher needs to write logs for each request. The connection between the dispatcher and system logs is another pipe-filter structure. In order to get the current time, the dispatcher should register to a clock subsystem to receive time value periodically. The connections between the clock and dispatcher can be considered as another publisher-subscriber structure. We model this TRMCS system in PAT. Due to the page limit, we can not present the complete model in the paper. Interested readers could refer to the more complete online version of the case study at http://www.comp.nus.edu.sg/~pat/adl/.

```
1. #assert TRMCS deadlockfree;
2. #assert TRMCS |= [](("r[0]:UserIssueRequest")
                        -> <>"p:RequestProcessed");
3. #assert TRMCS |= [](("r[0]:UserIssueRequest")
                       -> <>"p:OutputLog:putData");
4. #assert TRMCS |= [](("d:MainDatabaseCrash")
                       -> <>"b:BackupDBWorking");
```

A variety of interesting properties could be verified in terms of TRMCS architecture design. We pick four of them listed above for demonstration purposes. The first one is the deadlock-free analysis. This property is checked to guarantee the system never reaches a deadlock state. It is checked to be valid in our model. Properties 2, 3 and 4 are stated in LTL formulae. The second property means whenever a request is reported by receive port 0, it shall be processed by the dispatcher. This property guarantees that the system never misses any incident request, which is critical in medical systems. Similarly, property 3 states that the incident information must be recorded in the log whenever the dispatcher receives a request from receive port 0. The last property is used to check whether the control will be automatically switched to the backup database whenever the main database is unavailable, so as to guarantee the robustness of TRMCS system. The above three LTL properties are verified to be valid in our model. For more interesting properties, readers should refer to the case study web page online.

B. Implementation and Performance Evaluation

The Process Analysis Toolkit (PAT) [9], [14], [8] is designed to apply state-of-the-art model checking techniques for system analysis. It is a self-contained framework to support reachability analysis, deadlock-freeness analysis, full LTL model checking, refinement checking as well as a powerful simulator and verifier. PAT is designed to be an extensible and modularized framework [9] that allows users to build customized model checkers to support the analysis of different system notations. Currently, modules supporting modeling and verification of current systems [13], real-time systems [15], probabilistic systems [17], [16] are supported in PAT. The ADL module in PAT is dedicated to support the Wright# model analysis and verification. We conducted experiments on the Client-server, Pipe-filter, and TRMCS 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. We can see in the three examples, PAT performs reasonably well. It handles 10^7 states/transitions in a few hours. The first two examples extends the Client-server style and Pipe-filter style via defining multiple instances. In the last example, the number of states and running time increase rapidly as the parameters are enlarged. This is because of the TRMCS system is more complex in the interior communications. It is composed by 8 components and 8 connectors for the simplest one and embedded with vast of interleave and parallel operations. With the purpose of comparing the results of solely modeling in Wright# language with utilizing the style library, we designed two models to conduct comparison. The results demonstrate that the amount of user code is reduced nearly 1/3 via inheriting the style library. The main reason is that most connectors can be extended in user's model directly. For example, in TRMCS system, the connectors of Client-server, Pipe-filter and Publish-subscriber can be reused without any modification. Furthermore, inheriting styles from style library can improve the correctness of user model due to the dedicated verification of each architecture style in the library.

V. Conclusion

In this paper, we present a formal approach to the modeling and verification of software architecture designs using the PAT framework. We defined syntax and LTS operational semantics of the Wright# architecture description language. The extension supports both static and dynamic system behaviors modeling and configuration. Based on the formal semantics, we implemented a dedicated model checking module for Wright# in the PAT verification framework. The module – ADL supports automated verification and simulation of software architecture models in PAT. As complex system are often modeled using a multi-styled approach, we further developed an architecture style library which embodies a set of commonly used architecture style patterns to facilitate the

Model	Size	Property	States/Transitions	Time(Sec)
CS	C=4	deadlock-free	5841/20174	0.78
CS	C=5	deadlock-free	40365/173101	6.98
CS	C=6	deadlock-free	277857/1426404	58.85
CS	C=3	$\Box(Request : req \rightarrow \Diamond Provide : pro)$	3164/8508	0.4
CS	C=4	$\Box(Request : req \rightarrow \Diamond Provide : pro)$	29068/101358	4.28
CS	C=5	$\Box(Request : req \rightarrow \Diamond Provide : pro)$	250056/1078930	51.29
PF	P=5,F=6	deadlock-free	6144/21504	0.88
PF	P=6,F=7	deadlock-free	24576/98304	3.92
PF	P=7,F=8	deadlock-free	98304/442368	18.28
PF	P=5,F=6	$\Box(input : get \rightarrow \Diamond output : put)$	38913/142850	5.8
PF	P=6,F=7	$\Box(input : get \rightarrow \Diamond output : put)$	180225/753666	34.2
PF	P=7,F=8	$\Box(input : get \rightarrow \Diamond output : put)$	819201/3842050	246.8
TRMCS	H=1,M=1,R=1	deadlock-free	502079/2173895	119.69
TRMCS	H=2,M=1,R=1	deadlock-free	852407/3989713	221.79
TRMCS	H=1,M=1,R=1	$\Box(MainCrash \rightarrow \Diamond BackupWork)$	2359919/10039998	1337.73
TRMCS	H=2,M=1,R=1	$\Box(MainCrash \rightarrow \Diamond BackupWork)$	-	-

TABLE I
EXPERIMENT RESULTS

modeling process. The users can easily extend and reuse these verified structures in designing their customized systems. We demonstrate the effectiveness of our approach through a real-world case study of a Teleservices and Remote Medical Care System (TRMCS) modeling and verification. In addition, performance evaluations are presented to measure the scalability of the approach.

In the future, we plan to develop a Graphic User Interface (GUI) to assist the visual design of software architecture models in the PAT framework. The GUI should provide diagram representations of the architecture models and seemly connect to the model checking back-end for simulation and verification support. Another direction is to further develop on the language aspects of the Wright# notation, e.g., extending it with real-time and probabilistic properties to capture the quantitative time and uncertainty factors of components and connections in a software architecture description.

ACKNOWLEDGEMENT

This work is partially supported by the following projects: ASTAR SERC PSF 1121202016, MOE2009-T2-1-072, TRF Project "Research and Development in the Formal Verification of System Design and Implementation", and IDG31100105 / IDD11100102 from Singapore University of Technology and Design.

REFERENCES

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

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

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

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

[5] P. Inverardi, H. Muccini, D. Richardson, and S. Ficks. The Teleservices and Remote Medical Care System (TRMCS). IWSSD-10, 2000.

[6] 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.

[7] 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*, pages 70–80, New York, NY, USA, 2006. ACM.

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

[9] 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.

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

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

[12] 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.

[13] J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating Specification and Programs for System Modeling and Verification. In *Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 127–135. IEEE Computer Society, 2009.

[14] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[15] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying stateful timed csp using implicit clocks and zone abstraction. In K. Breitman and A. Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer, 2009.

[16] J. Sun, Y. Liu, S. Song, J. S. Dong, and X. Li. Prts: An approach for model checking probabilistic real-time hierarchical systems. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin / Heidelberg, 2011.

[17] J. Sun, S. Song, and Y. Liu. Model checking hierarchical probabilistic systems. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2010.

[18] S. Wong, J. Sun, I. Warren, and J. Sun. A Scalable Approach to Multi-Style Architectural Modeling and Verification. In *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*, pages 25–34. IEEE Press, 2008.