PAT approach to Architecture Behavioural Verification

Nacha Chondamrongkul*, Jing Sun†, Ian Warren‡
Department of Computer Science
The University of Auckland
Auckland, New Zealand
*ncho604@aucklanduni.ac.nz
† jing.sun@auckland.ac.nz
‡ i.warren@auckland.ac.nz

***Abstract*—Software architecture design plays a vital role in software development, as it gives an overview of how the software system should be constructed and executed at runtime. The verification of software architecture design is hence important but it is an error-prone task that heavily relies on knowledge and experience of the software architect, especially for a large software system that its behaviour is complex. Automated verification can be a solution to this problem, however, the specification language must be expressive enough to describe the behaviour of different design entities. This paper presents an enhancement of an architecture description language supported by PAT. The enhancement aims to improve the expressiveness of the language, in order to support the automated behaviour verification of software architecture design. With this enhancement, different behaviour of specific component and connector can be thoroughly checked and traced. The implementation of this enhancement is presented to demonstrate how the standard model checking engine such as PAT can be extended to support an architecture description language. We evaluated our approach with a case study and the result is presented.**

***Index Terms*—Software Architecture, Architecture Description Language, Model Checking, Linear Temporal Logic**

I. Introduction

Software architecture design gives an overview of how the software system is implemented and works. If the software architecture design is made incorrectly, it can cause the project to fail or delay due to design re-correction, therefore the verification is a significant task. However, the software architecture designs are usually represented by informal notations, such as graphical diagram and text. The design interpretation can hence be inconsistent and the verification process is an error-prone and time consuming task, even to those with extensive experience and knowledge. If the software architecture design can be formally defined, the verification task can be automated. Therefore, applying the formal methods to the software architecture design would be a useful approach to this problem.

Many architecture description languages (ADL) have been proposed to formally define the software architecture design model such as [1], [2], [3], and [4]. With the formal model in ADL, different properties can be defined and automatically verified with the model checker. Allen and Garlan [1] proposed Wright, an ADL that allows connections in software architecture design to be formally defined in communicating sequential process (CSP). The formal format of design model allows to check the architectural compatibility among connections. However, the behavioural property definition and verification had not been completely addressed so there are number of works that aim to fulfil this. Darwin [5] was proposed to allow behavioural properties to be defined in the linear temporal logic (LTL) and use LTSA [2] as a model checker to verify them. Defining and verifying behaviours in the evolving software system is a challenge. Oquendo [3] presented π-Method with an ADL based on π-calculus. The ADL for π-Method helps to formally define evolvable software system in both structural and behavioural view. In addition, the refinement model can be defined to check and preserve the behavioural properties.

Some works have applied process algebra to formalize specific behaviour in the software system because of its expressiveness in describing system behaviour. Aldini et al. [6] presented a guideline that includes a principle of formalizing system behaviour into process algebra. The manual formalization from the design model to ADL has been an obstacle to making it widely used by the software engineers, due to the fact that the majority of them do not have background knowledge in the formal methods and the verification output from model checker can be difficult to understand. Therefore, the degree of formality needs to be balanced with the practicality. Some approaches, such as Bose et al. [7], Baresi et al. [8] and CHARMY [4], hence provide a feature that translates the input model in graphical notation into formal language that can be automatically checked. While, some approaches, such as Arcade [9], aim to make the verification output from the model checker more readable. The graphical abstraction may promote the practicality and understandability of using formal ADL, but the ambiguity might occur from the lack of complete semantic mapping between the graphical input and the model checking input. In addition, most of the existing approaches use standard model checking engines that is not designed with the architecture design concept. For example, Wright uses FDR as a model checker, while Darwin and

DOI reference number:10.18293/SEKE2019-041

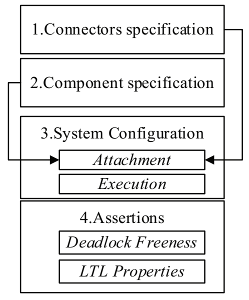


Fig. 1. Overall Approach

CHARMY uses LSTA and SPIN [10] respectively. As a result, the constructed state model is usually not optimized in term of understandability and scalability [11].

This work aims to enhance the expressiveness of Wright# [12], an architecture description language that is supported by PAT [13]. Wright# is an extension of Wright with the support of architecture styles reusing. This enhancement provides an expressive way to describe the execution of software system through system, component and connector specification. The overall approach can be found in Figure 1, which shows the specification that consists of four major parts. The connector can be specified according to the architecture styles. The component specification includes components involved in the system under design. In system configuration, the connector instances are created and attached to the component before we define how the system is executed through the process that initiates different components. The assertion is where the number of properties are defined for checking specific behaviour in the system. Deadlock freeness, a standard PAT feature, can also be used to verify the software architecture design. As the original Wright# produces a vague event labels that are difficult to trace. Therefore, the event labelling is optimized to clearly represent specific events in the connectors and components. As a result, the LTL properties can be defined to check the specific behaviours of design entities and the verification output is easier to trace back. The PAT extension module is developed as a graphical interface tool to support Wright# and its enhancement. In addition, we demonstrate the expressiveness of the design specification and property definition with a software architecture design of e-commerce software system.

The remainder of this paper is organized as follows. Section 2 explains the formalization of behaviour specification in the software architecture design. Section 3 presents the implementation of tool and how we develop a module in PAT framework to support Wright#. We demonstrate our approach with an e-commerce case study in Section 4. Section 5 concludes this paper and addresses the directions of future work.

II. Formalization of Architectural Design

In this section, we present the formalization of behaviour specification of software architecture design in Wright#. Wright# notation aims to define software architecture design in component and connector view. CSP, a process algebra notation, is used to formally describe the interactive behaviours of component and connector. We extend the PAT tool to support this ADL and transform it into native CSP that forms Labelled Transition System (LTS). With LTS, the desired behaviours of software system can be automatically checked through LTL, as well as the deadlock situation.

A. Formal Modelling

Wright# is an ADL that is inspired by Wright with four basic design entities namely the component, connector, port and role. The component represents a computational unit, while the connector represents linkage between the components. The connector can includes one or more roles representing how the communication works. The component contains a number of ports that can be attached to one or more roles defined by different connectors.

There are three parts of design model to be described in ADL namely connector definition, component definition and system configuration. Each definition of connector is corresponding to different type of communication according to the architecture style. The component are defined to represent actual component and port within the software system under design. The system configuration defines how component and connector are attached, as well as the execution process. These definition contains defined processes that are based on CSP. Table I shows the syntax of process expression that can be used to describe the processes within the software architecture design.

TABLE I
Process Expression Syntax

$e \rightarrow P$	Event prefixing
$ch!p \rightarrow P$	Channel output
$ch?p \rightarrow P$	Channel input
$P \parallel Q$	Parallel process
$P \mid\mid\mid Q$	Interleaving process
$P < \star > Q$	Coupling process
Stop	deadlock stop
Skip	terminate successfully

An event represents an abstract observation of a software system. It may refer to certain system state at a given time. Event prefixing hence represent a circumstance when an event e occurs then process P is executed next. Channel output and input are used to send and receive data from its executing environment respectively. Let ch be a channel and p is data to be sent or received; P is a process to be executed next. A pair of processes can be defined as parallel, interleaving and coupling. The coupling operator does not exist in native CSP or CSP# but it is added to the syntax to represent coupling process between components. Let P and Q be a process and $P < \star > Q$. When the process P is triggered, it contains a sequence of event that an event sequentially calls the process Q to execute and return back to where it is called on process P. In order to define coupling, process P must contain an event *process*, which is when the coupling process Q is called to execute.

1) Connector Definition: The connectors are firstly defined to manifest how roles interact together. The processes for role are defined with the sequence of events. The channel is used to represent communication between different roles, which results in transition between events. Below is a sample code that conveys the communication for the client-server structure. The channel *req* is used to make a request from the client to the server and channel *res* is used to return response message from the server to the client.

connector *CSConnector*{
  role $client(j) \ = \ request \rightarrow req!j \rightarrow res?j$
    $\rightarrow process \rightarrow client(j);$
  role $server() \ = \ req?j \rightarrow invoke$
    $\rightarrow process \rightarrow res!j \rightarrow server(); \ \}$

The connector for publisher-subscriber styles can be defined as shown below, where a channel *pub* is used to broadcast data from the publisher to the subscriber.

connector *PSConnector*{
  role $publisher(j) \ = \ process \rightarrow pub!j \rightarrow Skip;$
  role $subscriber() \ = \ pub?j \rightarrow process \rightarrow subscriber(); \ \}$

2) Component Definition: The component definition contains a set of port definition. Each port has a process defined as the sequence of event that the port performs internally within the component. The script below shows two sample component namely SPClient and SPServer. The *SPClient* component has a *test* port defined and the *SPServer* has *run* port defined.

component *SPClient* {
  port $test() \ = \ precheck \rightarrow output \rightarrow test(); \ \}$
component *SPServer* {
  port $run() \ = \ invoke \rightarrow execution \rightarrow run(); \ \}$

3) System Configuration: The system configuration contains details of how components interact among each other and can be defined as follow. Firstly, the instance of connector needs to be created with the *declare* statement based on a defined connector. Secondly, the ports of connector are attached to one of more roles of connector instances using *attach* statement. If more than one roles are attached, a process expression composed of multiple role and process operator can be defined. Lastly, the *execute* statement declares how the system are executed with a process expression.

system *SampleCS* {
  declare $cslink \ = \ CSConnector;$
  attach $SPClient.test() \ = \ cslink.client();$
  attach $SPServer.run() \ = \ cslink.server();$
  execute $SPServer.run() \parallel SPClient.test(); \ \}$

Careful readers may notice an event *process* defined at the role processes. This event triggers an execution of a process defined on the attached port. According to the sample system configuration shown above, the LTS is illustrated in Figure 2, which the events of port is shown in italic.

The *process* event also serves as the point of execution when the coupling process is defined. The coupling process may occur in many situation. For example, the multi-tier architecture that a tier can accept a request and consequently make a request to the upper tier. Another example is in Service-oriented architecture when a service is invoked and calls another service. Below is a sample of attached coupling role processes. The *run* port of *SPServer* is attached to a coupling of two roles from different client-server connectors namely *cslink* and *dlnk*, which calls the component *SPServer* and *DBServer* respectively where DBServer is a component. In this case, the role processes are nested within one another as LTS shown in Figure 3. This coupling feature eases the complexity of defining the coupling process in native CSP.

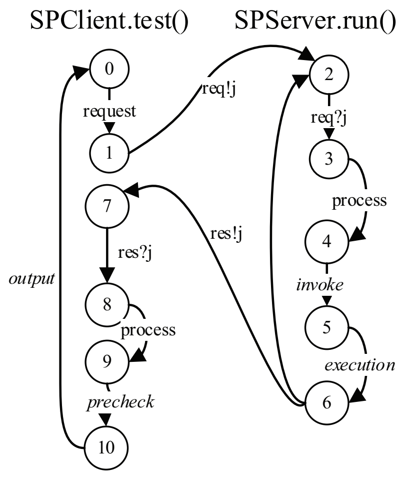


Fig. 2. LTS for the sample clien-server system

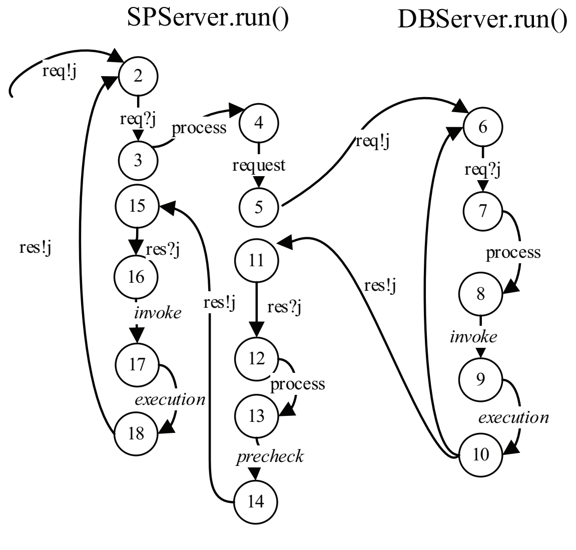


Fig. 3. LTS for the sample coupling process

attach $SPServer.run() \ = \ cslink.server() < * > dlnk.client();$
attach $DBServer.run() \ = \ dlnk.server();$

B. Behaviour Verification

After the software architecture design model is defined with ADL, different assertions representing the query about system behaviours can be defined. PAT supports a number of different assertion checking including linear temporal properties and deadlock freeness.

1) Deadlock Freeness: Deadlock is a situation when the software system can not progress further towards completion; so that the entire system halts and waits indefinitely. A well known scenario is when components wait for the mutual exclusive resource. In the software system, deadlock occurs when components call each other as circle, so the port that initializes the process loops back to itself. More concrete examples will be provided in the case study section. With the sample model explained in the previous section, a deadlock can be checked against a defined system using the *deadlockfree* statement as shown below.

assert *SampleSystem deadlockfree*;

2) Linear Temporal Properties: A full set of linear temporal logic is supported by PAT. Therefore, operators such as $\Box$ (always), $\Diamond$ (eventually), X (next), R (release) and U (until) can be included in the linear temporal logic defined for checking properties. Let F be a LTL formula, the assertion syntax for defining LTL properties is as follows.

assert *SampleSystem* $\models F$;

In order to support expressiveness of defining system behaviour, the property can be implicitly defined to check the behaviour of a specific component or connector.

Let *Comp* be any component, *Prt* is port of that component and *Evt* is one of the event defined in the port process. *F* is a LTL formula to check the behaviour of the component:

$F = [Comp.Prt.Evt] \mid \Box\ F \mid \Diamond\ F \mid X\ F \mid U\ F \mid R\ F$

Let *Comp* be any component, *Conn* be a connector, *Rle* be an attached role and *Evt* be one of the event defined in the role process. *F* is a LTL formula to check the behaviour of connector:

$F = [Comp.Conn.Rle.Evt] \mid \Box\ F \mid \Diamond\ F \mid X\ F \mid U\ F \mid R\ F$

For example, $\Box\Diamond SPServer.run.execution$ expresses a property to check if the *execution* event always eventually occurs at the *SPServer* component. $\Diamond DBServer.dblink.client.request$ expresses a property to check if the *request* event at attached client role of *DBServer* component eventually occurs.

III. Tool Implementation

To support editing architecture design model in ADL and automated behaviour verification, the PAT ADL module is developed by extending PAT framework. This module includes parser that helps to parse the ADL code into objects representing different entities of software architecture design model. The parser in the original PAT tool was developed using ANTLR version 3, where different parts of parsing code in C# are merged inside the grammar file. This style of development is difficult to make any extension and maintenance. Therefore, we adopt ANTLR version 4.0, where the source code of language parsing can be separated from the grammar file. The complete source code of PAT ADL can be found at https://bit.ly/2Vc855I.

The overall process performed by ADL module can be illustrated in Figure 4. The editor tool allows users to edit ADL file according to the syntax explained in the previous section. When the user makes a verification, the source code in ADL is processed by the PAT ADL in the background and return the result back to the user interface on the editor tool. We developed a grammar file based on the CSP# and used ANTLR to automatically generate a parser program. The parser program helps to read a source code in ADL and the visitor program is developed to convert different ADL statements into objects representing entities such as component, connector, system configuration, port, role, attachment and assertion. With *Specification* module, ADL objects are later automatically transformed into PAT native objects representing the CSP processes. The ADL to CSP transformation can be briefly explained as follows.

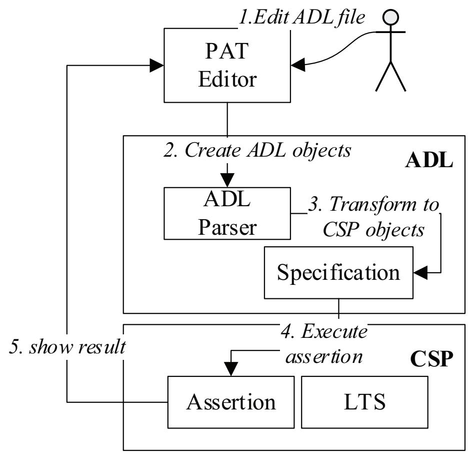


Fig. 4. Overall Process of the ADL module

- One process is defined corresponding to each attached role of a defined port.
- One process is defined to represent a defined port, which calls the attached role process according to the expression defined in the *attach* statement.
- One process is created for a defined system and call port processes according to expression defined in the *execute* statement.

For example, the sample client-server model explained in the previous section can be transformed into native CSP as shown below. Two channels namely *cslink_req* and *cslink_res* are defined for requesting and responding message. *SPClient_cslink_client* process is defined for the client role of *cslink*, and *SPClient_test* process is defined for the *test* port. Two processes namely *SPServer_cslink_server* and *SPServer_run* are defined in the same way for server. The *SampleSystem* process is defined to represent the main system process. The *LTS* sub-module helps to model LTS according to CSP and encapsulate the transition model. The transition model allows *Assertion* sub-module to traverse according to the depth-first search and breadth-first search algorithm, in order to make a verification. The verification result is displayed on the verification window of the editor tool.

```
channel cslink_ res 1;
channel cslink_ req 1;
SPClient_ cslink_ client(j) = (
SPClient_ cslink_ client_ request
→ cslink_ req!j → cslink_ res?j
→ (SPClient_ cslink_ client_ result
→ (SPClient_ cslink_ client_ process
→ (SPClient_ test_ precheck
→ (SPClient_ test_ output → SPClient_ cslink_ client(j))))));
SPClient_ test() = SPClient_ cslink_ client();
SPServer_ cslink_ server() = cslink_ req?j
→ (SPServer_ cslink_ server_ invoke
→ (SPServer_ cslink_ server_ process
→ (SPServer_ run_ invoke
→ (SPServer_ run_ execution
→ (SPServer_ cslink_ server_ return
→ cslink_ res!j → SPServer_ cslink_ server()))))));
SPServer_ run() = SPServer_ cslink_ server();
SampleSystem() = (SPServer_ run() || SPClient_ test());
```

IV. Case Study

We select a part of real-world e-commerce software system to demonstrate and evaluate the practicality of our approach. The software architecture design of this system is shown (as UML component diagram) in Figure 5. The software system allows user to browse catalogue of the products, order and make a purchase on-line through the web store or mobile store. When the users make a purchase, the order manager component keeps the record of order and fetch the product from the inventory though the inventory control component. The inventory control automatically locates the ordering product from the warehouse and send details to the shipping control component. The shipping control allows packaging officer to prepare shipping package and log the shipping package for courier.

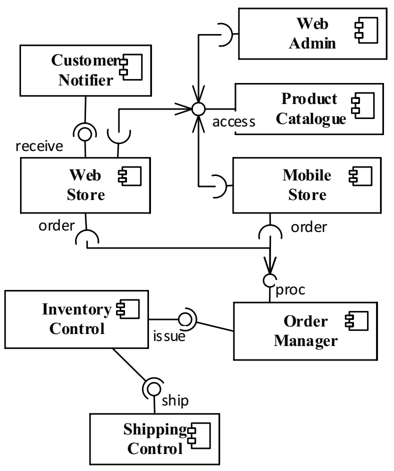


Fig. 5. Component diagram for E-commmerce system

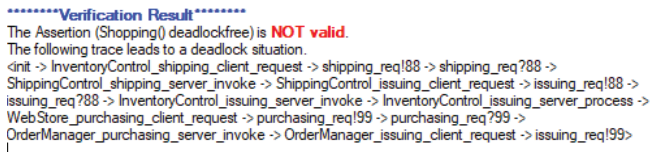
In this case study, we use the client-server and publisher-subscriber connectors as defined in the previous section. The model in ADL is partly shown below. The complete model can be found at https://bit.ly/2EwJGCg. The ports are defined according to the component diagram shown in Figure 5 along with the port processes that simulate what the components perform. Some port may omit the process details such as *ProductCatelogue* and *OrderManager* component. The system configuration of this system declares 4 client-server connector and 1 publisher-subscriber connector. The number of attachments are defined to link components together. The numbers passing in as a parameter for role processes represent the signature of data passing between the component and connector. The *proc* port of *OrderManager* is attached to two roles as a coupling process: *purchasing.server()* and *issuing.client()*. This is because the order can only be processed successfully when the inventory finished fetching the product, otherwise the order is rejected. The *purchasing.server()* represents the internal process for processing order within the *OrderManager* component, while the *issuing.client()* requests to the service on *InventoryControl* component to fetch the product. The *issue* port of *InventoryControl* component is attached to two roles as a parallel process. The *issuing.server()* represents the internal execution of *InventoryControl* component, while shipping.client() represents a request to manage the shipping details of the *ShippingControl* component. The *execute* statement defines all port processes to be executed in parallel. The execution can be modified to focus on checking some particular scenarios in response to the functionality of the software system, such as when the product is ordered and when the product catalogue is updated.

```
component WebStore {
    port browse() = render → output → browse();
    port order() = commit → email → order();
    port receive() = acknowledge → display → receive(); }
component MobileStore {
    port browse() = render → output → browse();
    port order() = commit → email → order(); }
component WebAdmin {
    port manage() = result → manage(); }
component CustomerNotifier {
    port alert() = promo → send → alert(); }
component InventoryControl {
    port issue() = locate → fetch → issue(); }
component ShippingControl {
    port ship() = inform → log → ship(); }
...
system Shopping {
  declare purchasing, issuing = CSConnector;
  declare shipping, cataccessing = CSConnector;
  declare newswire = PSConnector;
  attach WebStore.order() = purchasing.client(99);
  attach MobileStore.order() = purchasing.client(98);
  attach WebStore.receive() = newswire.subscriber();
  attach CustomerNotifier.alert() = newswire.publisher(77);
  attach WebStore.browse() = cataccessing.client(99);
  attach WebAdmin.manage() = cataccessing.client(98);
  attach ProductCatelogue.access() = cataccessing.server();
  attach OrderManager.proc() = purchasing.server()
                                  < * > issuing.client();
  attach InventoryControl.issue() = issuing.server()
                                  || shipping.client(88);
  attach ShippingControl.ship() = shipping.server();
  execute WebStore.order() || MobileStore.order()
     || WebStore.browse() || WebAdmin.manage()
     || ProductCatelogue.access() || OrderManager.proc()
     || InventoryControl.issue() || ShippingControl.ship();
```

Three assertions are defined as shown below. The first assertion helps to check if the software system design can leads to a deadlock. If the deadlock is found, the verification shows an invalid result with a counterexample, which gives a sequence of events leading how deadlock can occur.

$$\begin{aligned}&\textbf{assert}\ \textit{Shopping deadlockfree};\\ &\textbf{assert}\ \textit{Shopping} \models \Diamond \textit{WebStore.purchasing.client.process};\\ &\textbf{assert}\ \textit{Shopping} \models \Box(\textit{OrderManager.purchasing.server.process}\\ &\qquad\qquad \rightarrow \Diamond \textit{WebStore.order.email});\end{aligned}$$

With the system configuration above, the deadlock does not occur so it outputs a valid result. However, we can demonstrate when deadlock occurs by changing the attachment of the *ShippingControl.ship()* port to the *shipping.server()* $<*>$ *issuing.client()*. This makes the components call each other in a loop. The verification result can be seen in Figure 6. The event labels identify both component, connector, role and port that are involved in the deadlock. The second assertion makes use of the behaviour checking on the connector. It checks if the *process* event of *client* role is eventually triggered at the attached *purchasing* connector on the *WebStore* component. The third assertion combines the behaviour checking on both the component and connector, as it checks if every time the order is processed, the email will always be sent out to the customer. The results of these two LTL properties are valid. The verification statistic of these three assertions including number of states, number of transitions, total time usage and estimated memory usage can be found in Table II. As can be seen from the table, the number of visited states, memory usage and total time are relatively low.



```
********Verification Result********
The Assertion (Shopping() deadlockfree) is NOT valid.
The following trace leads to a deadlock situation.
<init -> InventoryControl_shipping_client_request -> shipping_req!88 -> shipping_req?88 ->
ShippingControl_shipping_server_invoke -> ShippingControl_issuing_client_request -> issuing_req!88 ->
issuing_req?88 -> InventoryControl_issuing_server_invoke -> InventoryControl_issuing_server_process ->
WebStore_purchasing_client_request -> purchasing_req!99 -> purchasing_req?99 ->
OrderManager_purchasing_server_invoke -> OrderManager_issuing_client_request -> issuing_req!99>
```

Fig. 6. Deadlock Result from PAT

TABLE II
VERIFICATION STATISTIC

Assertion	State#	Transition#	Time (sec)	Memory
Deadlock	25	24	0.0059828	8664 KB
LTL 1	16	24	0.0105143	8696 KB
LTL 2	159	278	0.0137552	42280 KB

V. CONCLUSION

We present an enhancement to Wright# ADL that supports the formal behaviour modelling in software architecture design. The language allows users to expressively define and verify the behaviour of components and connectors. The implementation of a PAT extension module to support Wright# and our enhancement is presented, in order to demonstrate how the standard model checker can be extended to support an ADL. We evaluate our approach with an e-commerce software system. Our approach can be used to clearly define the behaviour of different components and connectors in the design, as well as the interaction among them. The properties can be defined in LTL assertions to represent the desired system behaviour in response to the system functionalities. The deadlock analysis, a standard feature in PAT can be used. The event labels in a counterexample is informative enough to identify involved design entities that cause invalid behaviour. We found that the state space are relatively low but more evaluation need to be taken to prove the scalability.

For the future work, we plan to integrate this approach with other techniques such as ontology reasoning [14], in order to fulfil the semantics of the architecture design in the verification process. As the ontology representation is rich of semantic constrains that can help to verify and maintain the structure consistency in the design model before its behaviour is checked. More case studies in the real world could be used to evaluate the practicality and scalability of our approach. As the behaviours can be formally defined, it could be interesting to use it to detect the design smells or anti-pattern based on the system behaviours.

REFERENCES

[1] R. Allen and D. Garlan, "A formal basis for architectural connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, pp. 213–249, Jul. 1997.

[2] J. Magee and J. Kramer, *Concurrency: State Models & Java Programs*. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[3] F. Oquendo, "π-method: A model-driven formal method for architecture-centric software engineering," *ACM Sigsoft Software Engineering Notes*, vol. 31, pp. 1–13, 05 2006.

[4] P. Pelliccione, P. Inverardi, and H. Muccini, "Charmy: A framework for designing and verifying architectural specifications," *IEEE Transactions on Software Engineering*, vol. 35, pp. 325–346, 2009.

[5] J. Magree, "Behavioral analysis of software architectures using ltsa," in *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, May 1999, pp. 634–637.

[6] A. Aldini, M. Bernardo, and F. Corradini, *A Process Algebraic Approach to Software Architecture Design*. Springer Publishing Company, Incorporated, 2014.

[7] P. Bose, "Automated translation of uml models of architectures for verification and simulation using spin," in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, ser. ASE '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 102–.

[8] L. Baresi, C. Ghezzi, and L. Zanolin, *Modeling and Validation of Publish/Subscribe Architectures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 273–291.

[9] K. S. Barber, T. Graser, and J. Holt, "Providing early feedback in the development cycle through automated application of model checking to software architectures," in *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, Nov 2001, pp. 341–345.

[10] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[11] P. Zhang, H. Muccini, and B. Li, "A classification and comparison of model checking software architecture techniques," *Journal of Systems and Software*, vol. 83, no. 5, pp. 723 – 744, 2010.

[12] J. Zhang, Y. Liu, J. Sun, J. S. Dong, and J. Sun, "Model checking software architecture design," in *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*, Oct 2012, pp. 193–200.

[13] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness," ser. Lecture Notes in Computer Science, vol. 5643. Springer, 2009, pp. 709–714.

[14] N. Chondamrongkul, J. Sun, and I. Warren, "Ontology-based software architectural pattern recognition and reasoning," in *30th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, June 2018, pp. 25–34.