Proceedings of the International MultiConference of Engineers and Computer Scientists 2010 Vol II,
IMECS 2010, March 17 - 19, 2010, Hong Kong

A Rigorous Approach In Testing Service Component Architectures

Na Zhang, Xiaoan Bao, Zuohua Ding *

Abstract—**The paradigm of Service Oriented Architecture (SOA) has been gathering significant momentum in both academia and industry in recent years. SCA, which describes a model for building applications and systems using a SOA, extends and complements prior approaches to implementing services. However, it is very difficult to test if the service components integration satisfies the requirements. In this paper we propose a formal service component architecture model according to the specification of service component issued as a standard in Mar. 2007. A formulism, called Port Algebra, is developed to describe architecture. Test generation formulas are created to generate test cases for each service component. Based on the interactions of the components, SYN-sequences are derived to test the integration as a whole. Since the test generation is based on formulas, the test process could be in a systematical and automatical way.**

Keywords: Service component architecture, test case generation, SYN-sequence.

1 Introduction

The paradigm of Service Oriented Architecture (SOA) has been gathering significant momentum in both academia and industry in recent years. Web services technology provides a uniform framework to increase cross-language and cross-platform interoperability for distributed computing and resource sharing over the Internet. Furthermore, this paradigm of Web services opens a new cost-effective way of engineering software to quickly develop and deploy Web applications by dynamically integrating other independently published Web services as components to conduct new business transactions.

The essential feature of integration of services component model, among other aspects, poses new challenges to software quality. In a traditional software system, all of its components and their relationships are pre-decided before the software runs. Therefore, each component can be thoroughly tested, and the interactions among the components can be fully examined, before the system starts to execute. Service oriented architecture extends this paradigm by providing a more flexible approach to dynamically assemble distributed service components in an Internet-scale setting. However, how to test the integration of the service components remains a challenge. In fact, the flexibility of services-oriented computing is not without penalty since the value added by this new paradigm can be largely defeated if the selected service components do not thoroughly fulfill the requirements (i.e., functionally or non-functionally).

Thus we propose an approach to test service component and the composition based on the formal architecture model. Testing based on the architecture model can give us at least two benefits: 1) we can check if the system design satisfies the requirements, 2) by refining the test cases to the concrete test cases at code level[6] or by mapping the behavior traces to the action sequences, we can check the specification conformance.

Recently, several approaches have been proposed on architecture based testing. For examples, in [1], the SA dynamics was modeled by Labeled Transition System (LTS). In [7], the author developed a system abstraction with CSP in terms of testing. We know that LTS serves as the semantic model for languages such as CCS[5] and CSP[3]. Thus, roughly speaking, these two methods are LTS related. To handle the state space explosion problem generated from transition systems, [7] used hiding operator to provide a mechanism for removing particular events from the process' interface and making them internal and [1] derived a LTS abstraction from LTS.

We first create the test generation formulas for each component. Thereafter, based on the test cases of components, we create rules to generate SYN-sequences to test the whole architecture to see if the designed architecture satisfies requirements. Each SYN-sequence can cover some or all components.

The paper is organized as follows. In Section 2, we develop port algebra to describe component based software architecture. Section 3 builds formulas to generate test cases for components. Section 4 derives SYN-sequence to test whole architecture. The last section, Section 5, is the conclusion of this paper.

*The authors are with the Center of Math Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, Zhejiang 310018, P. R. China. E-mail: zouhuading@hotmail.com.

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

2 Service Component Architecture

Service component architecture (SCA) aims to encompass a wide range of technologies for service components and for the access methods which are used to connect them. One basic artifact of SCA is the service component, which is the unit of construction for SCA. Components are also the basic elements of business function in an SCA assembly, which are combined into complete business solutions by SCA composites.

2.1 Service Component

A service component corresponds to a service and it can be described by operation activities which represent well defined business functions. Component interactions are through data flowing from one activity to another. Service components are configured instances of implementations. Components provide and consume services via ports as shown in Fig.1.

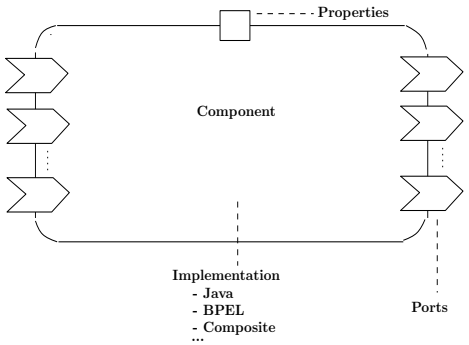

Figure 1: SCA component diagram

Port represents the addressable interfaces of the implementation and the requirement that the implementation has on a service provided by other components. Actually, each service offers the business functionality through ports. Each port has a Property which is defined as a message structure containing three members: *port type, partner link and input/output*. Once the port type and partner link have been assigned values, the property is configured. The configured property will i) influence the port by the port type and ii) lock the target service by the link. In other words, the properties of ports enable the configuration of an implementation.

In SCA, if a port provides a service, then it is called service port; If a port requires a service, then it is called reference port. We attempt to use port activities to describe component behavior. Motivated by the work of Böhm and Jacopini [2]: any sequential programs can be structured with sequences, branches and loops, we define some operations for ports. Assume that port p is in the form $p_{.m}$, where m is the associated message. The operators for ports are listed as follows:

1. ; Sequential Operator. To arrange the order of port acting. For example, expression $p_{1.m_1}; p_{2.m_2}$ means port p_1 acts before p_2.

2. $|_{cond}$ Choice Operator. Indicating choice relation. Expression $p_{1.m}|_{cond.m}p_{2.m}$ means if $cond.m = true$, execute ports p_1, otherwise execute p_2. If $cond = \emptyset$, then p_1 and p_2 may have same chance to act;

3. $[]_{cond}$ Loop Operator. Indicating that the port(s) inside of loop will be used continuously if $cond = true$, otherwise the loop will be broken. A special case is $cond = n$, where n could be a finite number or infinity. Expression $[p_{.m}]_{cond.m}$ means p will be called continuously if $cond.m = true$.

The above condition checking notation $cond.m$ is actually to check if the input/output of m satisfies the condition. To make all the notations simpler, as CSP, we may take message m away from the operations and just use the abstraction form: $p_1; p_2, p_1|_{cond}p_2, [p]_{cond}$, and the resulted ports are called: sequence, choice and loop, respectively.

Definition 2.1 *A sequential port is a combination of all kinds ports separated by ;.*

Definition 2.2 *A conceptual component is a set of finite sequential ports. Particularly, one sequential port is a conceptual component.*

Definition 2.3 *If all ports of a conceptual component are configured, then we say this component is configured.*

Definition 2.4 *A configured component is an implementation of its conceptual component.*

In practice, a configured component is an instance of conceptual component. One conceptual component can have a set of instances, each instance correspondence to an implementation manner, for example J2EE or C++.

2.2 Operations For Components

The SCA Assembly Model consists of a series of artifacts which define the configuration of an SCA domain in terms of composites which contain assemblies of service components and the connections and related artifacts which describe how they are linked together. SCA wires within a composite connect source component ports to target component ports(Fig. 2).

We consider that the wiring between two components are through message passing, where message passing are classified as Synchronous and Asynchronous. Synchronous message passing means that the sending port needs to

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

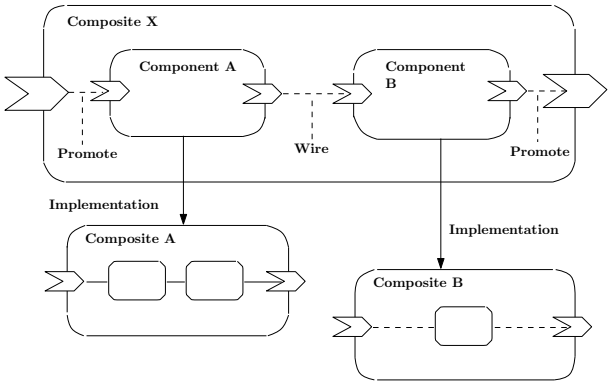

Figure 2: SCA composite diagram

wait until an acknowledgment is received. For example, a service port is called by-value by a client which is running in an operating system process different from that of the service itself (or clients running on different machines from the service). Asynchronous message passing means that the sending operation can proceed without waiting for the message to arrive at its destination. For example, a service port is called by-reference by clients that are running in the same process as the component that implements the service.

Thus the service port can be further classified as *synchronous-service* port and *asynchronous-service* port, reference port can be further classified as *synchronous -reference* port and *asynchronous-reference* port.

The wire is the link for the message passing between service port and reference port. If the message passing is from service port to service port or from reference port to reference port, then the link is called promote. Two types of wires and two types of promotes are defined for ports to build component links.

Two types of wires:

Wire I: from synchronous-service port to synchronous-reference port;
Wire II: from asynchronous-service port to asynchronous-reference port.

Two types of promotes:

Promote I: from (synchronous / asynchronous)-reference port to (synchronous / asynchronous)-reference port;
Promote II: from (synchronous / asynchronous)-service port to (synchronous / asynchronous)-service port.

The following notations are needed to support the above linkings. Let p_1 and p_2 be two ports,

1. $\bullet_m$. $p_1 \bullet_m p_2$ means service port p_1 sends message m to reference port p_2 and needs response back. $p_1 \bullet_m$ means service port p_1 sends message m out and does not need response back.
2. $\circ_m$. $p_1 \circ_m p_2$ means that reference port p_1 gets input message m from service port p_2 without providing response and $p_1 \circ_m (p_2)$ means that reference port p_1 gets input message m from service p_2 and provides response.
3. $\downarrow_m$. $p_1 \downarrow_m p_2$ means reference port p_1 passes message m to reference port p_2, and we say that p_2 is promoted by p_1.
4. $\uparrow_m$. $p_1 \uparrow_m p_2$ means service port p_1 passes message m to service port p_2, and we say that p_2 is promoted by p_1.

When one port is wired or prompted with another port, their port type and partner linker should match. For the convenience, the notation m is sometimes omitted. The follows are two examples of the usage of the wires.

Let $Comp_1$ and $Comp_2$ be two conceptual components:

$$Comp_1 : p_{11}; p_{12}; \ldots; p_{1i}; \ldots$$

$$Comp_2 : p_{21}; p_{22}; \ldots; p_{2j}; \ldots$$

Assume that port p_{1i} sends message m to port p_{2j} without requiring response, then we get a composite of $Comp_1$ and $Comp_2$ as:

$$p_{11}; p_{12}; \ldots; p_{1i-1}; p_{1i} \bullet_m; p_{1i+1}; \ldots$$

$$p_{21}; p_{22}; \ldots; p_{2j-1}; p_{2j} \circ_m; p_{2j+1}; \ldots$$

Assume that port p_{1i} sends message m to port p_{2j} requiring response back, then we get a composite of $Comp_1$ and $Comp_2$ as:

$$p_{11}; p_{12}; \ldots; p_{1i-1}; p_{1i} \bullet_m p_{2j}; p_{1i+1}; \ldots$$

$$p_{21}; p_{22}; \ldots; p_{2j-1}; p_{2j} \circ_m (p_{1i}); p_{2j+1}; \ldots$$

We call the above port notations *Port Algebra*. The syntax of a component can be represented as

$$\begin{aligned} comp \quad ::= \quad & p \,|\, p_1; p_2 \,|\, p_1|_{cond} p_2 \,|\, [p]_{cond} \\ & |\, p_1 \bullet p_2 \,|\, p_1 \circ p_2 \,|\, p_1 \uparrow p_2 \,|\, p_1 \downarrow p_2. \end{aligned}$$

2.3 Framework of Integration

The above discussion focuses on that one component is linked with another component through one pair of ports. This discussion can be extended to the situation that one component is linked with many components. Let $Comp$ be the conceptual model of a component that has all its

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

ports linked. To represent the links between *Comp* and other components, every port of *Comp* is attached by wire notation ($\bullet_m$, $\circ_m$) or promote notation ($\uparrow_m$, $\downarrow_m$).

Let C_1 and C_2 be two components. We use $\oplus$ and $\otimes$ to denote wire and promote operations between these two components, respectively. Then we have the following results. The proofs can be easily implied from component's sequential representation and thus omitted.

Theorem 2.5 *(associativity of $\oplus$) Let C_1, C_2 and C_3 be three components, then*

$$(C_1 \oplus C_2) \oplus C_3 = C_1 \oplus (C_2 \oplus C_3).$$

Theorem 2.6 *(associativity of $\otimes$) Let C_1, C_2 and C_3 be three components, then*

$$(C_1 \otimes C_2) \otimes C_3 = C_1 \otimes (C_2 \otimes C_3).$$

Corollary 2.7 *Let $Comp_1, Comp_1, \ldots, Comp_n$ be finite components, then the composite of these components are still components.*

Definition 2.8 *Let C be the set of all components. $\mathcal{A} =< C, \oplus, \otimes >$ is called an architecture.*

We have the following properties about the architectures.

Properties:

(1) **Closeness**: The compositions of component to component, component to architecture, architecture to architecture are still architectures.

(2) **Hierarchy**: An architecture can be decomposed to more architectures.

(3) **Expansibility**: A new component can be added to an architecture.

To save space, we omit the operational semantics of port activities.

3 Test Generation Formulas For Components

Usually, we regard test data generation as the process to partition input space to domains, and thus a test case is a sequence of input messages and an output message with some domains. Hence, in simulation, given the ordered input with some domains, the component will return some output. Since we do not know the inside state change and the exact information attached to a port in the middle of the process, for a port p, we use $I(p)$ to represent its input information, and $O(p)$ to represent its output information. I and O are thus defined as the associated messages of p. When the process migrates from one port to another port, all the information will be passed from one port to another port. If the process reaches an *Asynchronous Service* port, then the O information of this port will contain all the required input information for the output at this port, in other words, we obtain a test case.

To calculate the O information, we need some defines. Symbols $\wedge$ and $\vee$ mean AND and OR. To save notations, from now on, the condition notation *cond* also represents a set in which all elements must satisfy this condition and symbol ";" is also used to separate inputs. We also define

$$\begin{aligned}
&I(p_1; p_2) = I(p_1),\\
&I(p_1|_{cond} p_2) = I(p_1) \quad \text{or} \quad I(p_2),\\
&I([p_1 ...]_{cond}) = I(p_1),\\
&I(p_1 \circ) = I(p_1) \wedge I(o),\\
&I(p_1 \bullet) = I(p_1),\\
&I(p_1 \bullet p_2) = I(p_1) \wedge I(p_2),\\
&I(p_1 \uparrow p_2) = I(p_1),\\
&I(p_1 \downarrow p_2) = I(p_1).
\end{aligned}$$

For a port p, its O information is usually associated with its state, i.e, $O(p) \wedge \underline{p}$, which means that p reaches the exit state $\underline{p}$ and has an out information $O(p)$. Thus, for each port $\underline{p}$, following the semantics of the program language in the last section, we may define its O information as the result of the state change initiated by its input information:

$$O(p) = I(p) \wedge \overline{p}.$$

Based on this definition and the semantics, we can infer the following formulas for ports.

Proposition 3.1 *Sequence.* $O(p_1; p_2) = O(p_2)$.

Proposition 3.2 *Choice.* $O(p_1|_{cond} p_2) = (cond_1 \wedge O(p_1)) \vee (\neg cond \wedge O(p_2))$.

Proposition 3.3 *Loop.* $O([p_1; ...; p_2]_{cond}) = (cond \wedge O(p_2); \neg cond) \vee (I(p_1) \wedge^{\neg} cond)$.

Corollary 3.4 *If cond=n, then*

$$\begin{aligned}
& O([p_1; \ldots; p_2]_n)\\
= \; & O([p_1; \ldots; p_2]_1); \ldots; O([p_1; \ldots; p_2]_n)\\
= \; & O(p_1[1]); \ldots; O(p_2[1]); \ldots; O(p_1[n]); \ldots O(p_2[n])),
\end{aligned}$$

where $[p_1; \ldots; p_2]_i$ is the ith execution of $p_1; \ldots; p_2$ and $p[i]$ is the ith execution of p.

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Proposition 3.5 *Output.* $O(p\bullet) = O(\bullet)$.

Proposition 3.6 *Input.* $O(p\circ) = I(p); O(\circ)$.

Proposition 3.7 *Linking in synchronization.* $O(p_1 \bullet p_2) = I(p_1); O(p_2)$.

Comparing Propositions 3.6 and Proposition 3.7, we find that for a port requiring response back, this port actually needs an input. Thus, we only need to consider two types of ports: reference port that needs input, i.e., $p\circ$ and service port that sends output, i.e., $p\bullet$.

Since $O(\circ)$ and $O(\bullet)$ can be further computed as $O(\circ) = O(\circ_m) = m$ and $O(\bullet) = O(\bullet_m) = m$, the O information at each *asynchronous service* port would be a sequence of input messages and an output message separated by ";", or a sequence of input "conditioned messages" and an output message separated by ";", where "conditioned message" is $m \wedge cond$. For example, if p_1 is a reference port associated with message m_1 and p_2 is a service port associated with message m_2, then

$$\begin{aligned} O(p_1\circ_{m_1}; p_2\bullet_{m_2}) &= O(p_2\circ_{m_2}) \\ &= I(p_2); m_2 \\ &= O(p_1); m_2 \\ &= I(p_1); O(o_{m_1}); m_2 \\ &= I(p_1); m_1; m_2. \end{aligned}$$

Since each message corresponds to a port, a sequence of messages corresponds to a sequence of ports. In general, if a sequence of messages is, say, $m_1 \wedge cond_1; m_2 \wedge cond_2; \ldots; m_n$, then the corresponding sequence of ports is $p_{1.m_1\wedge cond_1}; p_{2.m_2\wedge cond_2}; \ldots; p_{n.m_n}$, where m_n is an output message associated with p_n and all other m_i's are input messages associated with other p_i's.

Hence, a test case can also be expressed by a sequence of reference ports, synchronous service ports and an asynchronous service port separated by ";". We may define our test coverage criteria: all test cases must cover each port at least once. Since a component is composed of reference ports and service ports, we may visualize that the component is the overlap of the test cases.

For a component C, we use $T(C)$ to denote the set of all test cases.

4 Forming Test Cases For Architectures

Service oriented architecture is composed of components that interact to each other through message passing, in other words, the behavior of one component will be affected by the message from other components. Thus, to make a test run, we may need test data from more than one component. In the execution, these data wind through control flow and interact to form events. Thus an event sequence can be used to guide the execution. Since we already have test cases for components, we will use the interactions among components to corporate these test cases to build event sequences.

Now, let C_1, C_2 and C_3 be three components. Assume their test sets are $T(C_1), T(C_2)$ and $T(C_3)$, respectively. Let $t_1 \in T(C_1)$ be the test case corresponding to an output. t_1 needs input x from C_2 at the input point p_{11}. Let $t_2 \in T(C_2)$ be the test case that passes the output x to t_1. Assume again that t_2 needs input y from C_3 at the input point p_{21} and let t_3 be the test case of C_3 that passes the output y to t_2. Then these test cases can be expressed as:

$$\begin{aligned} t_1 &: \ldots; p_{11}\circ_x; \ldots; p_{12}\bullet_d, \\ t_2 &: \ldots; p_{21}\circ_y; \ldots; p_{22}\bullet_x, \\ t_3 &: \ldots; p_{31}\bullet_y . \end{aligned}$$

Based on Proposition 3.6 and 3.7, we may combine these test cases by substituting the required input by the O information from its partner. We use $*_?$ to represent this combination operation, where the question mark will be replaced by the input message. For example,

$$\begin{aligned} & t_1 *_{\bar{x}} t_2 \\ = \; & (\ldots; p_{11}\circ_x; \ldots; p_{12}\bullet_d) *_x (\ldots; p_{21}\circ_y; \ldots; p_{22}\bullet_x) \\ = \; & \ldots; p_{11} \circ (\ldots; p_{21}\circ_y; \ldots; p_{22}\bullet_x); \ldots; p_{12}\bullet_d \end{aligned}$$

We use $T(C_1) * T(C_2)$ to denote all these combinations. Thus we can combine t_1, t_2 and t_3 together to form a new test case t: $t \in T(C_1) * T(C_2) * T(C_3)$,

$$\begin{aligned} & t \\ = \; & t_1 *_x (t_2 *_y t_3) \\ = \; & (\ldots; p_{11}\circ_x; \ldots; p_{12}\bullet_d) *_x \\ & \quad ((\ldots; p_{21}\circ_y; \ldots; p_{22}\bullet_x) *_y (\ldots; p_{31}\bullet_y)) \\ = \; & (\ldots; p_{11}\circ_x; \ldots; p_{12}\bullet_d) *_x \\ & \quad ((\ldots; p_{21} \circ (\ldots; p_{31}\bullet_y); \ldots; p_{22}\bullet_x) \\ = \; & \ldots; p_{11} \circ ((\ldots; p_{21} \circ (\ldots; p_{31}\bullet_y); \ldots; p_{22}\bullet_x); \\ & \quad \ldots; p_{12} \bullet_d . \end{aligned}$$

Since no other inputs are needed, we can complete the test based on these three components.

If we consider the above port activities as *events*, we may extract *event sequences* from the combined test cases. In this example, we have events:

$$\begin{aligned} e_1 &= p_{11}.x, \\ e_2 &= p_{12}.d, \\ e_3 &= p_{21}.y, \\ e_4 &= p_{22}.x, \\ e_5 &= p_{31}.y \end{aligned}$$

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Let $e_1 * e_2$ denote the interaction (synchronization) between events e_1 and e_2. Then the above test case has another from:

$$\begin{aligned}
&\{\\
&\quad E_1 = \{\ldots; e_1; \ldots; e_2\},\\
&\quad E_2 = \{\ldots; e_3; \ldots; e_4\},\\
&\quad E_3 = \{\ldots; e_5\},\\
&\quad (e_1 * e_4,\ e_3 * e_5)\}\\
&\},
\end{aligned}$$

where E_1, E_2 and E_3 are totally ordered sets. A name, SYN-sequences, is given to such kind of event sequences.

Definition 4.1 *[4] The SYN-sequence Q exercised by a concurrent execution is defined as a tuple $(Q_1, Q_2, \ldots, Q_n; \phi)$, where Q_i is the totally ordered sequence of sending and receiving events that occurred on a thread(process) or a synchronization object and ϕ is the set of synchronization pairs exercised in the execution.*

From the above discussion, for each $t_1 \in T(C_1)$, we can build a SYN-sequence, moreover, the input data is also obtained. From now on, when we say SYN-sequence, we always mean that the sequence has input data attached. Thus $T(C_1)$ has a set of SYN-sequences Q_{C_1}. Generally, if a service architecture has n components $C_1, C_2, \ldots, C_n$, then we will have n set of SYN-sequences $Q_{C_1}, Q_{C_2}, \ldots Q_{C_n}$. Since each C_i has finite test cases, each Q_{C_i} will have finite SYN-sequences. Among all Q_{C_i}, some SYN-sequence may be overlaped.

One should notice that in the definition, ϕ may contain pairs such as $(e_1 * e_2)$ and $(e_1 * e_3)$. This means that the input event e_1 may have 2 partners at the same time, in other words, e_2 and e_3 have the same chance to pass data to e_1. ϕ may also contain the pairs such as $(e_1 * e)$ and $(e_2 * e)$. This means that e_1 and e_2 all require e as the partner and they have same chance to contact e.

5 Conclusion

This paper presented a formal method to test SCA based design. A formulism, called Port Algebra, was developed to describe architecture. We construct a service component model according to the specification of service component issued as a standard in Mar. 2007. Our description was scalable since it did not require extra operators to build hierarchy structure. The architecture description was test based since it could be visualized as the overlap of all test cases. Since the test generation was based on formulas, the test process could be in a systematical and automatical way. Gas Station example was employed to demonstrate our point to some level. Our method can be used to test if the designed architecture satisfies requirements. Comparing with the existing methods, our method can avoid state explosion problem since the architecture model is not state based and all inside state information is omitted.

Acknowledgments

This work is partially supported by NSF of China under No.90818013, and Zhejiang NSF under Grant No.Z1090357.

References

[1] A.Bertolino P. Inverardi and H. Muccini, Formal methods in testing software architectures, SFM'03, *Lectures Notes in Computer Science* 2804, pp.122-147, 2003.

[2] C. Böhm and G. Jacopini, "Flow diagram, turing machines and languages with only two formation rules", *Communications of ACM*, vol.9, no. 5, pp.366-371, 1966.

[3] C.A.R. Hoare, Communicating Sequential Processes. Prentice Hall, 1985.

[4] Y. Lei and R. H. Carver, Reachability Testing of Concurrent Programs, *IEEE Transactions on Software Engineering*, vol. 32, no. 6, pp.382-403, June 2006.

[5] R. Milner, Communication and concurrency, Prentice Hall, 1989.

[6] H. Muccini, A. Bertolino and P. Inverardi, Using Software Architecture for Code Testing, IEEE Transactions on Software Engineering, vol. 30, no. 3, pp.160-171, 2004.

[7] S. Schneider, Abstraction and Testing, FM'99, LNCS 1708, pp. 738-757, 1999.

ISBN: 978-988-18210-4-1
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)