A Scheme for Dynamic Detection of Concurrent Execution of Object-Oriented Software*

Huo Yan Chen, Yu Xia Sun
Department of Computer Science, Jinan University, Guangzhou 510632, P. R. China

T. H. Tse
Department of Computer Science and Information Systems, The University of Hong Kong, Hong Kong

Abstract — *Program testing is the most widely adopted approach for assuring the quality and reliability of software systems. Despite the popularity of the object-oriented programs, its testing is much more challenging than that of the conventional programs. We proposed previously a methodology known as TACCLE for testing object-oriented software. It has not, however, addressed the aspects of concurrency and non-determinism.*

In this paper, we propose a scheme for dynamically detecting and testing concurrency in object-oriented software by executing selected concurrent pairs of operations. The scheme is based on OBJSA nets and addresses concurrency and non-determinism problems. An experimental case study is reported to show the effectiveness of the scheme in detecting deadlocks, race conditions and other coherence problems. The scheme supplements our previous static approach to detecting deadlock in Java multithreaded programs.

Keywords: Object-oriented program testing, dynamic detection and testing, concurrency, OBJSA net

1. Introduction

Object-oriented paradigm is becoming the main methodology for software systems analysis and design. The testing of object-oriented software, however, is more complex and difficult than that of conventional programs.

Various approaches to testing object-oriented software systems have been proposed [5, 6, 8, 9, 10, 11]. For example, we proposed in [6] a methodology *TACCLE* to test object-oriented software system at the class and cluster levels. We also presented in [5] an approach for statically detecting object-oriented software system at the method level. These earlier results, however, did not cater for concurrent or non-deterministic situations. Because of the popularity of Java and its strong multi-thread mechanisms, the dynamic testing of concurrency and non-determinism in object-oriented software systems is of increasing importance and should be addressed properly.

Carver and Tai [4] proposed to use sequencing constraints for specification-based testing of concurrent programs. Despite the effectiveness of the approach, the sequencing constraints only specified preceding and succeeding events in the concurrent system under test. They did not express other requirements and properties of the system. Zhu and He [12] proposed several adequacy criteria for testing concurrent systems based on high-level Petri nets and also proved subsumption relationships among them. They did not, however, provide techniques for constructing test cases to cover all or part of the criteria in [12].

In this paper, we propose a scheme for *dynamically* detecting and testing concurrency in object-oriented software by executing selected concurrent pairs of operations. Our scheme is based on OBJSA-net/CLOWN specifications [1, 2], which have been successfully used in a large and significant project proposed by the Italian electricity company ENEL.

We shall present the background concepts of OBJSA nets in the next section. We shall then discuss our proposed scheme in the subsequent sections.

2. Background Concepts

* This research is supported in part by the National Natural Science Foundation of China under Grant #60173038, the Guangdong Province Science Foundation under Grant #010421, and the Research Grants Council of Hong Kong.

To lay the foundations of the paper, we present in this section the basic concepts of OBJSA nets originally proposed in [1, 2]. We shall adhere as much as possible to the notation of [2] for the ease of understanding and comparison.

A *net* is a triple $N = (P, T, F)$, where P, T, and F are finite non-empty sets such that $P \cap F = \emptyset$ and $F \subseteq (P \times T) \cup (T \times P)$. The elements of P, T, and F are known as *places*, *transitions*, and *arcs*, respectively. In general, places are used to model conditions or system resources, and transitions are used to model operations or actions.

Let $V = P \cup T$ be the set of *vertices* of N. For any $v \in V$, ${}^{\circ}v = \{y \mid y \in V \wedge (y, v) \in F\}$ is called the *pre-set* of v, and $v^{\circ} = \{y \mid y \in V \wedge (v, y) \in F\}$ is called the *post-set* of v.

An *extended SA net* is a tuple $N = (P, T, F, W, \Pi)$, where (P, T, F) is a net. Places in P are partitioned into two disjoint classes OP and CP. The elements of OP are called *open places* and those of CP are called *closed places*. Transitions in T are partitioned into two disjoint classes OT and CT. The elements of OT are called *open transitions* and those of CP are called *closed transitions*. An arc $f \in OF \subseteq (OP \times OT) \cup (OT \times OP)$ is said to be an *open arc*. An arc $f \in CF \subseteq (CP \times T) \cup (T \times CP)$ is said to be *closed*. $W: F \rightarrow Nat$ is the *arc weight* function, where *Nat* denotes the set of natural numbers. In particular, $W(f) = 1$ for every open arc f. Π is a partition of P into disjoint classes $\Pi_1, \Pi_2, \ldots, \Pi_m$ such that every Π_i contains either open places only or closed places only, and for every $t \in T$, $\Sigma_{p \in (\Pi_i \cap {}^{\circ}t)} W(p, t) = \Sigma_{p \in (\Pi_i \cap t^{\circ})} W(t, p)$.

An extended SA net N is said to be *closed* if $OP = OT = \emptyset$, and *open* otherwise. The nets generated only by classes in CP are called *elementary subnets* of N.

Given an extended SA net $N = (P, T, F, W, \Pi)$ and an *algebraic specification* $SPEC = (S, Opt, Eq)$, an *OBJSA component* is a SPEC-inscribed net $(N, ins, SPEC)$ with an *initial marking* (or initial state) M_0, where $ins = (\varphi, \lambda, \eta)$ is a SPEC-inscription of N such that:

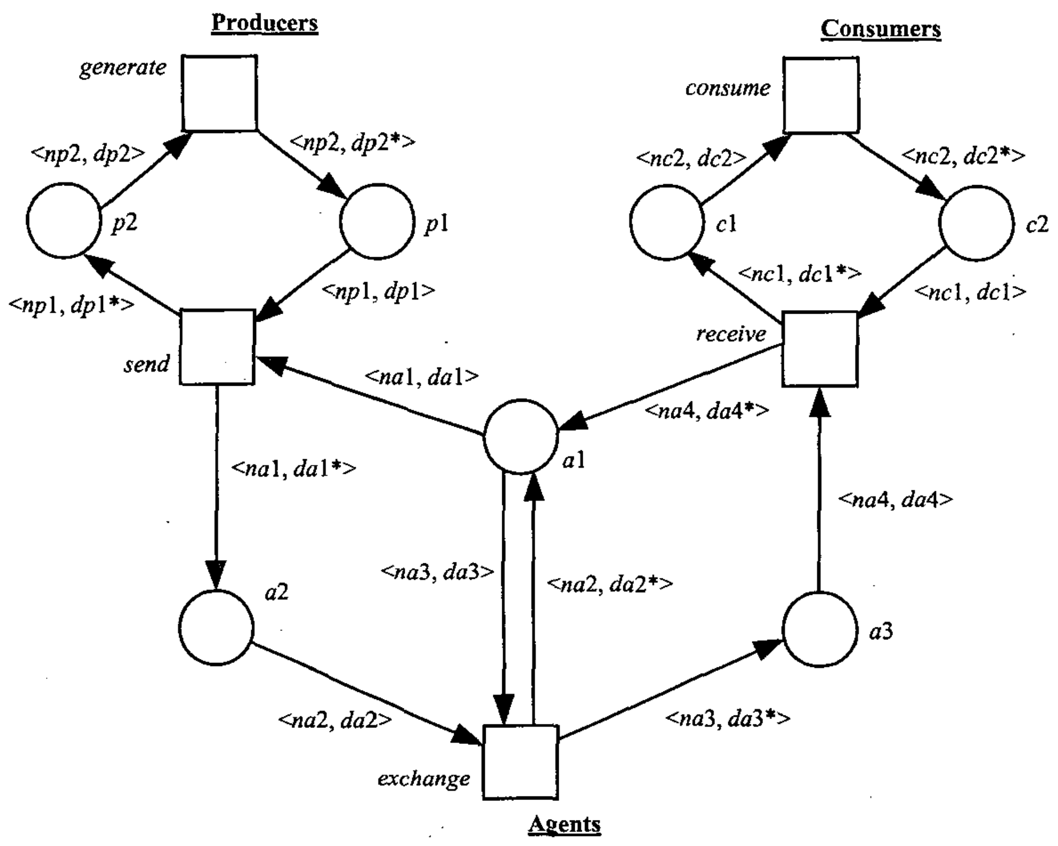
(a) φ: $P \rightarrow S$ is a *sort assignment* function, which divides places into various *sorts* (or object classes) while respecting the partition Π. Each element of sort $\varphi(p)$ is known as a *token*. It is of the form $<n_{i,j}, d_{i,j}>$, where $n_{i,j} \in Nt$ denotes the name of the token and $d_{i,j} \in D$ denotes its data content.

(b) λ: $T \rightarrow \Sigma_S$ is a φ-respecting *arc labeling* function, which assigns labels to the arcs surrounding every transaction as follows: For every $t \in T$, let ${}^{\circ}t = \{p_1, p_2, \ldots, p_a\}$ and $t^{\circ} = \{q_1, q_2, \ldots, q_b\}$. For every arc $f = (p_i, t)$, if f is open, its label is a variable $x_{i,1}$ of sort $\varphi(p_i)$; otherwise its label is of the form $x_{i,1} <+> x_{i,2} <+> \ldots <+> x_{i,W(f)}$, where each $x_{i,j}$ is a variable of sort $\varphi(p_i)$. Let X_t be a list of variables that label the input arcs of t. For every arc $f = (t, q_k)$, if f is open, its label is a term $y_{k,1}(X_t)$; otherwise its label is of the form $y_{k,1}(X_t) <+> y_{k,2}(X_t) <+> \ldots <+> y_{k,W(f)}(X_t)$, where each $y_{k,j}(X_t)$ is a term of sort $\varphi(q_k)$. Furthermore, for each variable $x_{i,j} = <n_{i,j}, d_{i,j}>$ in X_t, there exists a unique term $y_{k,r}(X_t) = <n_{k,r}{*}, d_{k,r}{*}>$ of sort $\varphi(q_k)$ such that $n_{k,r}{*} = n_{i,j}$ and $d_{k,r}{*} = \sigma_t(\ldots, d_{i,j}, \ldots)$ for some function σ_t that specifies the change of the data content due to the transition t.

(c) η: $T \rightarrow Bool$ is an inscription function that assigns to every transaction t a pre-condition $\eta(t, X_t)$ for firing it.

M_0 associates with each closed place p a multi-set of tokens of sort $\varphi(p)$, under the condition that if the name of a token appears in the marking of a place, it must not appear in the marking of any other place of the same elementary component. An open place $op \in OP$ is associated with all the possible terms of the sort $\varphi(op)$.

An OBJSA net is constructed in a bottom-up manner. An OBJSA component is said to be *elementary* if the underlying net N contains only one elementary subnet. An OBJSA component is said to be *open* if the underlying net N is open. They are constructed by composing elementary or other open components together. An *OBJSA net* is a *closed* OBJSA component, formed by composing elementary or open OBJSA components, such that the underlying net N is closed. Details of composition rules can be found in [2].



*np*1, *np*2, *nc*1, *nc*2, *na*1, *na*2, *na*3, *na*4: *ObjectName* in the form [*type*: *Type*, *id*: *Nat*], where *Type* denotes the set of object sorts, *Nat* denotes the set of natural numbers, $type \in \{p, c\}$, *p* denotes producer, *c* denotes consumer, and *id* denotes the object identifier.

*da*1, *da*2, *da*3, *da*4, *da*1*, *da*2*, *da*3*, *da*4*: *FullMessage* in the form [*msg*: *Message*, *dest*: *Nat*], where *dest* denotes a destination object. $da1^* = dp1$. $da2^* = nullmsg$. $da3^* = da2$. $da4^* = nullmsg$.

*dp*1, *dp*2, *dp*1*, *dp*2*, *nullmsg*: *FullMessage*. $dp1^* = nullmsg$. $dp2^* = produceMessage(np2)$.

*dc*1, *dc*2, *dc*1*, *dc*2*, *null*: *Message*. $dc1^* = msg(da4)$. $dc2^* = null$.

*pr*1, *pr*2, *pr*3: *Bool*. $pr1 = (type(na1) == p) \wedge (id(na1) == id(np1))$.

$pr2 = (type(na2) == p) \wedge (type(na3) == c) \wedge (dest(da2) == id(na3))$.

$pr3 = (type(na4) == c) \wedge (id(na4) == id(nc1))$.

Figure 1. OBJSA Net Specifying the System in Example 1

Given an OBJSA component with a transition $t \in T$ such that ${}^\circ t = \{p_1, p_2, \ldots, p_a\}$ and $t^\circ = \{q_1, q_2, \ldots, q_b\}$, a *firing mode* for t is an assignment function $\beta_t: X_t \to T_{\varphi(p_i)}$ that associates a ground term of sort $\varphi(p_i)$ to every variable $x_{i,j}$ in the list X_t.

Given a firing mode β_t, for each place $p \in P$, $IN(p, t)$ is defined as follows: (a) $IN(p, t) = \{\beta_t(x_{i,j}) \mid j = 1, 2, \ldots, W(p, t)\}$ if $p = p_i \in {}^\circ t \cap CP$; (b) $IN(p, t) = \{\beta_t(x_{i,1})\}$ if $p = p_i \in {}^\circ t \cap OP$; (c) $IN(p, t) = \varnothing$ otherwise. Given a marking M, a transition $t \in T$ and a firing mode β_t, we say that t is β_t-*enabled at* M if, for each place $p_i \in {}^\circ t$, $IN(p_i, t) \subseteq M(p_i)$ and $\eta(t, \beta_t) = true$.

An enabled transition at the initial marking M_0 of an OBJSA net is called a *source transition*. Starting with M_0, the number of times that a source transition can be consecutively fired is known as the *index* of the source transition. A source transition with the largest index is called the *greatest source transition*.

In order to help readers understand OBJSA net concepts, we give here an example adapted from [2]. After we have discussed our scheme in Sections 3 and 4, we shall revisit this example in an experimental case study in Section 5.

Example 1. Suppose a system comprises 5 producers and 4 consumers asynchronously exchanging messages through a network of 5 + 4 agents. The constituents of the system can be specified by OBJSA open components *Producers*, *Consumers*, and *Agents*, respectively. The OBJSA net specifying the system is shown in Figure 1. Its initial marking M_0 is as follows:

$M_0(p2) = \{<[p, 1], nullmsg>, <[p, 2], nullmsg>, <[p, 3], nullmsg>, <[p, 4], nullmsg>, <[p, 5], nullmsg>\}$;

$M_0(c2) = \{<[c, 1], null>, <[c, 2], null>, <[c, 3], null>, <[c, 4], null>\}$;

$M_0(a1) = \{<[p, 1], nullmsg>, <[p, 2], nullmsg>, <[p, 3], nullmsg>, <[p, 4], nullmsg>, <[p, 5], nullmsg>, <[c, 1], null>, <[c, 2], null>, <[c, 3], null>, <[c, 4], null>\}$;

$M_0(p) = \varnothing$ for $p \in P - \{p2, c2, a1\}$.

3. Our Scheme

This section describes our scheme for dynamically detecting and testing concurrency in object-oriented software by simultaneously executing selected concurrent pairs.

Given an OBJSA net *Osn*, let M_i be a reachable marking. If two transitions t_{i1} and t_{i2} (or their corresponding operations) can be fired simultaneously at M_i, we say that t_{i1} and t_{i2} are a *concurrent pair* at M_i. In formal terms, t_{i1} and t_{i2} are a concurrent pair at M_i if and only if (t_{i1} and t_{i2} are enabled respectively) and $\neg\exists p\ (p \in {}^\circ t_{i1} \cap {}^\circ t_{i2} \wedge \forall \beta_t\,(IN(p, t_{i2}) \not\subset M_i(p) - IN(p, t_{i1})))$. For simplicity, let $I1$, $I2$, and M denote $IN(p, t_{i1})$, $IN(p, t_{i2})$, and $M_i(p)$, respectively. It can easily be proved that if $I1 \subseteq M$ and $I2 \subseteq M$, then (a) $I2 \not\subset M - I1$ implies $I1 \cap I2 \neq \varnothing$ and (b) $I2 \not\subset M - I1$ if and only if $I1 \not\subset M - I2$. Hence, the expression $IN(p, t_{i2}) \not\subset M_i(p) - IN(p, t_{i1})$ above can be replaced by $IN(p, t_{i1}) \not\subset M_i(p) - IN(p, t_{i2})$. In other words, t_{i1} and t_{i2} will be a concurrent pair at M_i if and only if (t_{i1} and t_{i2} are enabled respectively) and $\neg\exists p\ (p \in {}^\circ t_{i1} \cap {}^\circ t_{i2} \wedge \forall \beta_t\,(IN(p, t_{i1}) \not\subset M_i(p) - IN(p, t_{i2})))$.

Let τ be a sequence of individual or concurrent transitions. The notation $M_i \xrightarrow{\tau} M_j$ means that, starting with the marking M_i, we can consecutively fire the transitions in τ to obtain the marking M_j. When τ is *null*, $M_i = M_j$. The notation $M_i \xrightarrow{t_{i1} \| t_{i2}} M_j$ means that simultaneously firing t_{i1} and t_{i2} at M_i will obtain the marking M_j. In fact, $M_i \xrightarrow{t_{i1} \| t_{i2}} M_j$ can be taken as a test case.

If $M_0 \xrightarrow{\tau_i} M_i$, where M_0 is the initial marking of *Osn*, we say that M_i can *be reached* by τ_i. Thus, the test case $M_i \xrightarrow{t_{i1} \| t_{i2}} M_j$ can be written as $\tau_i \bullet t_{i1} \| t_{i2}$, which means that we can reach M_j if we start from M_0, consecutively fire the individual or concurrent transitions in τ_i, and then simultaneously fire the concurrent pair t_{i1} and t_{i2}.

Let t_0 be the greatest source transition of a given OBJSA net *Osn*. Our scheme for selecting concurrent test cases of the form $M_i \xrightarrow{t_{i1} \| t_{i2}} M_{i+1}$, or $\tau_i \bullet t_{i1} \| t_{i2}$, contains the following steps:

```
(1) set $M_c := M_0$, $T_c := \{t_0\}$, $\tau_c := null$, and $i := 1$;
(2) if there is a sequence $(t_0, t_1, \ldots, t_k)$ of transitions in
        $T_c$ (where $k < |T_c|$) such that, starting with $M_c$,
        we can reach $M_i$ after firing $t_0 \bullet t_1 \bullet \ldots \bullet t_k$, that
        is, $M_c \xrightarrow{t_0 \bullet t_1 \bullet \ldots \bullet t_k} M_i$, and if we can find $t_{i1}$
        ($\notin T_c$) and $t_{i2}$ ($\in T_c$) that can be fired
        simultaneously at $M_i$,
    then {
        $M_i \xrightarrow{t_{i1}\|t_{i2}} M_{i+1}$, $T_c := T_c \cup \{t_{i1}\}$, $\tau_i := \tau_c \bullet t_0$
            $\bullet\, t_1 \bullet \ldots \bullet t_k$, and $\tau_c := \tau_i \bullet t_{i1}\|t_{i2}$;
        return a test case $\tau_i \bullet t_{i1}\|t_{i2}$;
        };
    else exit from the scheme;
(3) $i := i + 1$;
    if we can find $t_{i1}$ ($\notin T_c$) and $t_{i2}$ ($\in T_c - \{t_0\}$) that
        can be fired simultaneously at $M_i$,
    then {
        $M_i \xrightarrow{t_{i1}\|t_{i2}} M_{i+1}$, $T_c := T_c \cup \{t_{i1}\}$, $\tau_i := \tau_c$,
            and $\tau_c := \tau_c \bullet t_{i1}\|t_{i2}$;
        return a test case $\tau_i \bullet t_{i1}\|t_{i2}$;
        go to (3);
        };
    else set $M_c := M_i$ and go to (2);                    □
```

4. Discussions

We presented in [7] an approach to detecting deadlocks in Java multithreaded programs. It constructs Calling Hierarchy Diagrams and Lock-Calling-Suspend Diagrams from the programs under test, and then analyzes special properties to determine whether is there are potential deadlocks in the programs. The approach is *static* and *white-box based.* It cannot, for instance, find deadlocks due to dynamic binding.

As a supplement to the approach described in [7], the scheme introduced in the last section of this paper is for detecting and testing concurrency in object-oriented software by executing selected concurrent pairs of operations. It is *dynamic* and *black-box based.* It can detect deadlocks due to dynamic binding.

Because of non-determinism in concurrent programs, we must use *replay techniques* to execute each test case $\tau_i \bullet t_{i1}\|t_{i2}$ obtained in the proposed scheme. Details of replay techniques can be found in [3].

Our approach can expose various errors due to concurrency, such as deadlocks, race conditions, and other coherence problems. The scheme can be applied not only to Java programs, but also to programs of other languages that support concurrency.

5. Experimental Case Study

Applying the above scheme to Example 1, we obtained the following test cases:

$$\tau_1 \bullet t_{11}\|t_{12} = t_0 \bullet t_1\|t_0;$$
$$\tau_2 \bullet t_{21}\|t_{22} = t_0 \bullet t_1\|t_0 \bullet t_2\|t_3;$$
$$\tau_3 \bullet t_{31}\|t_{32} = t_0 \bullet t_1\|t_0 \bullet t_2\|t_3 \bullet t_3\|t_2;$$
$$\tau_4 \bullet t_{41}\|t_{42} = t_0 \bullet t_1\|t_0 \bullet t_2\|t_3 \bullet t_3\|t_2 \bullet t_4\|t_3;$$

where t_0 = generate, t_1 = send, t_2 = exchange, t_3 = receive, and t_4 = consume.

We implemented a Java system consisting of 5 producers and 4 consumers as specified in Example 1, and then injected deadlocks, race conditions and other coherence problems into the program. All the injected faults were revealed by our approach.

6. Conclusion

We have presented a black-box and dynamic scheme for detecting and testing concurrency in object-oriented software by executing selected concurrent pairs of operations based on OBJSA-net specifications. An experimental case study has also been reported. More case studies and experiments will be conducted as future research.

Acknowledgement

We would like to express our thanks to Shuang Quan Li for the implementation and experiments in the case study.

References

[1] Battiston E., A. Chizzoni, and F. D. Cindio. CLOWN as a testbed for concurrent object-oriented concepts. In *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, pages 131–163. *Lecture Notes on Computer* Science, Springer, Berlin, 2001.

[2] Battiston E., F. de Cindio, and G. Mauri. Modular algebraic nets to specify concurrent systems. *IEEE Transactions on Software Engineering*, 22 (10): 689–705, 1996.

[3] Carver R. H. and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8 (3): 66–74, 1991.

[4] Carver R. H. and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24 (6): 471–490, 1998.

[5] Chen H. Y. The design and implementation of a prototype for data flow analysis at the method-level of object-oriented testing. In *Proceedings of the 2002 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2002*), pages 140–145. IEEE Computer Society Press, Los Alamitos, California, 2002.

[6] Chen H. Y., T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10 (1): 56–109, 2001.

[7] Chen H. Y. Race condition and concurrency safety of multithreaded object-oriented programming in Java. In *Proceedings of the 2002 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2002*), pages 134–139. IEEE Computer Society Press, Los Alamitos, California, 2002.

[8] Doong R.-K. and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3 (2): 101–130, 1994.

[9] Kung D. C., J. Z. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *Proceedings of the 19th Annual International Computer Software and Applications Conference* (*COMPSAC '95*), pages 239–244. IEEE Computer Society Press, Los Alamitos, California, 1995.

[10] Smith M. D. and D. J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5 (3): 45– 53, 1992.

[11] Turner C. D. and D. J. Robson. A state-based approach to the testing of class-based programs. *Software: Concepts and Tools*, 16 (3): 106–112, 1995.

[12] Zhu H. and X. He. A theory of testing high-level Petri nets. In *Proceedings of* the International *Conference on Software: Theory and Practice, 16th IFIP World Computer Congress*, pages 443–450. Beijing, China, 2000.