A Strategy for Selecting Synchronization Sequences to Test Concurrent Object-Oriented Software[1,2]

Huo Yan Chen
Department of Computer Science
Jinan University, China
tchy@jnu.edu.cn

Yu Xia Sun
Department of Computer Science
Jinan University, China
sabrinna@21cn.com

T. H. Tse[3]
Department of Computer Science
and Information Systems
The University of Hong Kong
tse@csis.hku.hk

Abstract

Testing is the most commonly used approach to the assurance of software quality and reliability. The testing of object-oriented software is much more complex than that of conventional programs. Although we proposed previously a method called TACCLE for testing object-oriented software at the class and cluster levels, it did not cover concurrent or non-deterministic situations.

This paper puts forward a strategy for selecting synchronization sequences to test concurrent object-oriented software, including non-deterministic cases. It is based on OBJSA net/CLOWN specifications. Experiments have been carried out in a case study to verify the efficacy of the strategy.

Keywords: *Object-oriented program testing, concurrency, non-determinism, OBJSA net*

1. Introduction

Object-oriented programming is among the most popular software development technique today. It consists of many distinct features within the same paradigm. The application domain is modeled by objects that include not only data structures but all the operations also. Data abstraction and encapsulation allows us to separate object behaviors and interfaces from implementation details. Inheritance enhances reuse. Polymorphism enables us to use the same operation symbols for different purposes in different situations. The resulting software has been found to be more flexible, maintainable, and reusable.

On the other hand, objects may interact with one another in unforeseen combinations in object-oriented systems, which are much more complex to test than the hierarchical structures in conventional programs. Various proposals for testing object-oriented software system have been made [1–5]. In [1], for instance, we recommended a *TACCLE* methodology for testing object-oriented software at the class and cluster levels. We must concede, however, that it did not take concurrency and non-determinism into account.

Since the introduction of the Java language with strong multi-thread mechanisms and Internet support, the static analysis and dynamic testing of concurrency and non-determinism are increasing in importance. Although Carver and Tai [6] presented an effective technique for the use of sequencing constraints for specification-based testing of concurrent programs, these constraints are only limited to preceding and succeeding events. Although Zhu and He [7] put forward adequacy criteria for testing concurrent systems based on high-level Petri nets and proved subsumption relationships among them, the authors did not offer techniques for actually constructing test cases.

In [8], Chen presented an approach for the *static* analysis of concurrent and non-deterministic object-oriented programs in Java. As a supplement, we shall present in this paper a strategy for selecting synchronization sequences to *dynamically* test non-deterministic concurrent object-oriented software. The strategy is based on OBJSA net / CLOWN specifications [9, 10]. In a

[3] **Corresponding author**.

companion paper [11], we shall further supplement this strategy by proposing a scheme for dynamically executing selected pairs of synchronization sequences.

An OBJSA net in CLOWN is composed of a super-posed automata (SA) net inscribed with algebraic terms of an OBJ module. It supports the components-based analysis and incremental development of specifications with good modularity and reusability. By adding OBJ notions to Petri nets, such as *order-sorted algebra*, *theory*, and *view* [12], OBJSA net provides testers with more information for the selection of test cases. An *OBJSA Net-support Environment* (*ONE*) has been built to facilitate construction and validation of the specifications.

We shall outline in Section 2 the underlying concepts and rules of OBJSA net specifications. Based on these fundamentals, we shall propose in Section 3 a strategy for selecting synchronization sequences to test concurrent non-deterministic object-oriented software. Section 4 presents an experimental case study to verify the effectiveness of the strategy. Section 5 concludes the paper.

2. Underlying Concepts of OBJSA Nets

In this section, we outline the underlying concepts of OBJSA nets, which were due originally to Battiston et al. Interested readers may refer to [9, 10] for more details.

(a) Nets

A net is a bipartite graph with two types of nodes: places and transitions. Places are used to model the statuses of system conditions and transitions are used to model operations that affect such statuses. More formally, a *net* is a triple (P, T, F) such that:

(i) P is a finite non-empty set of *places*,
(ii) T is a finite non-empty set of *transitions,*
(iii) $P \cap T = \varnothing$
(iv) $F \subseteq (P \times T) \cup (T \times P)$ is a set of *arcs*.

$V = P \cup T$ is known as the set of *vertices*. For any vertex $v \in V$, the set of arcs $^{o}v = \{x \in V \mid (x, v) \in F\}$ is called its *pre-set*, and the set of arcs $v^{o} = \{y \in V \mid (v, y) \in F\}$ is called its *post-set*.

(b) Extended SA Nets

Given a net (P, T, F), an *extended SA net N* is a tuple $(P, T, F, W, \prod)$ satisfying the following conditions:

(i)   *OP* is a set of *open places* and *CP* is a set of *closed places* such that $OP \cup CP = P$ and $OP \cap CP = \varnothing$.
(ii) *OT* is a set of *open transitions* and *CT* is a set of *closed transitions* such that $OT \cup CT = T$ and $OT \cap CT = \varnothing$.

(iii) $OF \subseteq (OP \times OT) \cup (OT \times OP)$ is a set of *open arcs*.
(iv) $CF \subseteq (CP \times T) \cup (T \times CP)$ is a set of *closed arcs*.
(v) $W: F \rightarrow Nat$ is an *arc weight* function. It assigns to every arc a natural number denoting its weight. In particular, every open arc is assigned a weight of 1.
(vi) $\prod$ is a partition of *P*, dividing it into *m* subsets $(\prod_i)_{i=1,2,...,m}$ such that $\prod_1 \cup \prod_2 \cup ... \cup \prod_m = P$ and $\prod_i \cap \prod_j = \varnothing$ for $i \neq j$. Each $\prod_i$ contains open places only or closed places only. For every transition $t \in T$, the total weights of all the arcs from the places in $\prod_i$ to t is the same as the total weights of all the arcs from t to the places in $\prod_i$.

An extended SA net is *open* if it contains at least one open place or open transition; otherwise it is *closed*. An extended SA net is *elementary* if it contains only closed places.

(c) OBJSA Nets

Let $N = (P, T, F, W, \prod)$ be an extended SA net and $SPEC = (S, \Sigma, E)$ be an algebraic specification, where *S* is a set of *sorts* (or object classes), Σ is a family of operation symbols, and *E* is a set of equational axioms, possibly conditional. An *OBJSA component* is a SPEC-inscribed net (*N*, *ins*, *SPEC*), where *ins* = (φ, λ, η) satisfies the following conditions:

(i) $\varphi: P \rightarrow S$ is a *sort assignment* function. It classifies the places into different sorts while respecting the partition $\prod$. Every element of sort $\varphi(p)$ is called a *token*. It is of the form $\langle n, d \rangle$, where *n* denotes the token name and *d* denotes the data content.

(ii) $\lambda: F \rightarrow Term_{S,\Sigma}[X]$ is an *arc labeling* function. It assigns labels to the arcs while respecting the sort assignment. Given any transition $t \in T$ and any place $p \in {}^{o}t$, the label of the arc $f = (p, t)$ is the concatenated string $x_1 \oplus x_2 \oplus ... \oplus x_{W(f)}$. Each x_i is a variable of sort $\varphi(p)$. Let X_t be a list of variables that label the input arcs of *t*, and *X* be the set of variables for all the arcs of the net. Given any transition $t \in T$ and any place $q \in t^{o}$, the label of the arc $f = (t, q)$ is the concatenated string $y_1(X_t) \oplus y_2(X_t) \oplus ... \oplus y_{W(f)}(X_t)$. Each $y_j(X_t)$ is a term of the form $\langle n_j', d_j' \rangle$ of sort $\varphi(q)$ such that $n_j' = n_i$ and $d_j' = \sigma_t(..., d_i, ...)$ for some $x_i = \langle n_i, d_i \rangle$ in X_t and for some operation σ_t that specifies the change of the data content due to the transition *t*.

(iii) $\eta: T \rightarrow 2^X \rightarrow Bool$ is a *guard* function that assigns a pre-condition $\eta(t)(X_t)$ to every transaction *t* before it can be fired.

Furthermore, an OBJSA component is associated with an *initial marking* (or initial state) M_0, which assigns to each closed place *p* a family of tokens of sort $\varphi(p)$.

5

An OBJSA component is *elementary* if the underlying net N contains only an elementary subnet. An OBJSA component is *open* if it is formed by composing elementary or other open components, such that the underlying net is open. Finally, an *OBJSA net* is a *closed* OBJSA component, formed by composing elementary or open OBJSA components, such that the underlying net is closed.

(d) Firing Mode

Given an OBJSA component, a *firing mode* for a transition $t \in T$ is an assignment function $\beta_t: X_t \rightarrow Term_{S,\Sigma}$ that substitutes every variable $x \in X_t$ by a *ground term* (that is, a term without variables) of sort $\varphi(p)$ some $p \in {}^o t$.

For a place $p \in P$, a transition $t \in T$, and a firing mode β_t, if $p \in {}^o t$, then $IN(p, t)$ is defined as $\{\beta_t(x_i) \mid i = 1, 2, ..., W(p, t)\}$; otherwise $IN(p, t)$ is defined as the empty set. Similarly, if $p \in t^o$, then $OUT(t, p)$ is defined as $\{\beta_t(y_j(X_t)) \mid j = 1, 2, ..., W(t, p)\}$; otherwise $OUT(t, p)$ is defined as the empty set. Given a marking M, a transition $t \in T$, and a firing mode β_t, if $\eta(t)(\beta_t(X_t)) = true$ and $IN(p, t) \subseteq M(p)$ for every place $p \in {}^o t$, then t is said to be β_t-*enabled at* M. In this case, t may fire in mode β_t. Such firing returns a new marking M' such that $M'(p) = M(p) \setminus IN(p, t) \cup OUT(t, p)$ for every $p \in P$.

3. Our Strategy for Selecting Synchronization Sequences

Based on the concept of OBJSA nets, we present in this section a strategy for selecting synchronization sequences to test non-deterministic concurrent object-oriented software.

Given an OBJSA net *Osn*, we say that a marking M is *reachable* if, starting from the initial marking M_0, we can obtain M by firing a sequence τ of consecutively enabled transitions. In this case, we call τ an *enabled sequence* and write $M_0 \xrightarrow{\tau} M$. We assume that τ is annotated with arc information so that any intermediate marking can be tracked unambiguously. If $\tau = null$, $M = M_0$.

Let $M^* = \{M \mid M_0 \xrightarrow{\tau} M$ for some enabled sequence $\tau\}$ be the set of reachable markings of *Osn*. For any $M \in M^*$, the number of non-deterministic enabled transitions of *Osn* at M is called the *enabled degree* of *Osn* at M, and is denoted by $ed(Osn, M)$. Let $md(Osn) = \max\{ed(Osn, M) \mid M \in M^*\}$ be the *maximum non-deterministic enabled degree*. We say that M_g is a *goal marking* of *Osn* if $ed(Osn, M_g) = md(Osn)$. Let $ET(M_g)$ be the set of enabled transitions at M_g. Obviously, $|ET(M_g)| \geq md(Osn)$.

$md(Osn)$ must exist, because $ed(Osn, M)$ for any $M \in M^*$ must be a non-negative integer satisfying $ed(Osn, M) \leq |T|$. The goal marking M_g corresponding to $md(Osn)$, however, is not necessarily unique. At M_g, the firings of transitions in *Osn* have the maximum non-determinism and hence the maximum competition on system resources. At this moment, the system is in a most complex state and is therefore most error prone. Hence, we should catch the state corresponding to M_g in our testing. This is the motivation and the general idea of our strategy for selecting synchronization sequences.

Suppose $M_0 \xrightarrow{\tau_g} M_g$. Let $TC_g = \{\tau_g \cdot t_i \mid t_i \in ET(M_g)\}$, where $\tau_g \cdot t_i$ denotes the firing of t_i immediately after τ_g. We say that τ_g is a *goal sequence* and TC_g is a *goal set*. Our strategy is to select TC_g as a set of initial test cases.

Let TC be any set of test cases. For any given transition $t \in T$, is there an $M \in M^*$ such that t is enabled at M? The problem is difficult because M^* is infinite in general, and is NP-hard even if M* is finite.

From the complexity point of view, seeking a goal marking M_g, goal sequence τ_g, or a goal set TC_g is also a difficult problem. For a particular *Osn*, however, we can give a heuristic strategy for seeking a *ballpark goal marking* M_g' and the corresponding *ballpark goal sequence* τ_g' such that there will be as many enabled transitions at M_g' as possible. The corresponding *ballpark goal set* TC_g' is $\{\tau_g' \cdot t_i \mid t_i \in ET(M_g')\}$.

An enabled transition at initial marking M_0 is called a *source transition* of *Osn*. The number of times that a source transition in an enabled sequence can be consecutively fired from M_0 is known as the *index* of the source transition. We shall refer to a source transition with the largest index as the *greatest index source transition*, or *GIST* for short.

Suppose t_0 is the GIST of *Osn* with M_0. Our heuristic strategy to seek a ballpark goal marking M_g' and the corresponding ballpark goal sequence τ_g' includes a reachability search [13–17], such as the following hill-climbing procedure:

(1) set $\tau_g' = null$ and $M_g' = M_0$;

(2) fire t_0 at M_0 and obtain M_1, that is, $M_0 \xrightarrow{t_0} M_1$;

(3) at M_1:
   if t_0 is not enabled, return τ_g' and M_g', and exit;
   if t_0 is still enabled, {
      if there is no other enabled transition t_1, return τ_g' and M_g', and exit;
      if there is another enabled transition t_1, {

set $\tau_g' = \tau_g' \cdot t_0$, $M_1' = M_1$, $M_g' = M_1$, and $i = 1$;
                };
        };

(4) if there are *adequately many* [4] non-deterministic
    enabled transitions at M_g', then return τ_g' and M_g',

    and exit; otherwise $M_i' \xrightarrow{\quad t_i \quad} M_{i+1}$;

(5) at M_{i+1}:
    if t_0 is not enabled, return τ_g' and M_g', and exit;
    if $t_0, t_1, \ldots, t_i$ are still enabled, {
        if there is no other enabled transition t_{i+1}, return
            τ_g' and M_g', and exit;
        if there is another enabled transition t_{i+1}, {
            set $\tau_g' = \tau_g' \cdot t_i$, $M_{i+1}' = M_{i+1}$, $M_g' = M_{i+1}'$, and
                $i = i+1$;
            go to (4);
            };
        };
    if t_0 is still enabled but some of $t_1, t_2, \ldots, t_i$ are not
    enabled, {
        if firing some of $t_0, t_1, \ldots, t_{i-1}$ several times can
        enable all of $t_0, t_1, \ldots, t_i$, {
            let τ_i be the corresponding sequence of the fired
                transitions and let M_{i+1}' be the marking

            obtained, that is, $M_{i+1} \xrightarrow{\quad \tau_i \quad} M_{i+1}'$;

            if there is no other enabled transition t_{i+1} at M_{i+1}',
                return τ_g' and M_g', and exit;
            if there is another enabled transition t_{i+1} at M_{i+1}', {
                set $\tau_g' = \tau_g' \cdot t_i \cdot \tau_i$, $M_g' = M_{i+1}'$, and $i = i + 1$;
                go to (4);
                };
            };
        else return τ_g' and M_g', and exit;
        };

In order to understand the strategy, readers may like to
construct a decision tree for the above steps.

After obtaining the ballpark goal sequence τ_g' and the
corresponding M_g', we can construct the ballpark goal set
$TC_g' = \{\tau_g' \cdot t_i \mid t_i \in ET(M_g')\}$ and take TC_g' as a set of test
cases.

A transition in OBJSA net represents an operation in
the implementation triggered by some message passing
from another object. A test case in TC_g' represents a
sequence of inter-object operations in the implementation,
which we shall call a *synchronization sequence*. We can
then substitute the relevant parts of these synchronization

---

sequences into the axioms of corresponding classes, and
apply our class-level testing strategy for observational
equivalence [1]. Because of the non-determinism of
concurrent programs, we must use a *replay technique* to
execute each synchronization sequence in TC_g'. Details of
replay techniques can be found, for example, in [18].

Note that TC_g' may not cover all the transitions of *Osn*.
In this case, and if TC_g' cannot reveal errors, we also need
to construct other sets of test cases covering the remaining
transitions $T \setminus (ET(M_g') \cup \{t \mid t$ appears in $\tau_g'\})$ as
supplements. Finally, the testing of *Osn* may be repeated
by choosing other initial markings. Such initial markings
may be given in the specification, or obtained from a
reachability search or category-partitioning scheme.

5. Case Study and Experiments

As a case study, we have applied the above strategy to
a system consisting of four generators and five users
synchronously exchanging messages through four buffers.
These three constituents can be specified by OBJSA open
components *Generator*, *User*, and *Buffer*, respectively.
Each object is identified by a natural number together with
the component type. The structures of these components
are similar to those in the worked example in [9]. The
OBJSA net specifying the integrated system fused from
these components is shown in Figure 1. Suppose an initial
marking M_0 is as follows:

$M_0(g2) = \{\langle[g, 1], null\rangle, \langle[g, 2], null\rangle,$
$\qquad \langle[g, 3], null\rangle, \langle[g, 4], null\rangle\}$;

$M_0(u2) = \{\langle[u, 1], nullmsg\rangle, \langle[u, 2], nullmsg\rangle,$
$\qquad \langle[u, 3], nullmsg\rangle, \langle[u, 4], nullmsg\rangle,$
$\qquad \langle[u, 5], nullmsg\rangle\}$;

$M_0(b1) = \{\langle[b, 1], nullmsg\rangle, \langle[b, 2], nullmsg\rangle,$
$\qquad \langle[b, 3], nullmsg\rangle, \langle[b, 4], nullmsg\rangle\}$;

$M_0(p) = \varnothing$ for $p \in P \setminus \{g2, u2, b1\}$.

Following the definition in Section 4, we obtain the GIST
$t_0 = generate$. Following the proposed strategy, we have
the following process: (i) Set $\tau_g' = null$ and $M_g' = M_0$. (ii)
Fire t_0 and obtain M_1 by the rules in Section 2. (iii) Here t_0
is still enabled. By the definition in Section 2, another
transition $t_1 = fetch$ is enabled. Set $\tau_g' = generate$, $M_1' = M_1$,
$M_g' = M_1$, and $i = 1$. (iv) Fire $t_1 = fetch$ and obtain M_2 by the
rules in Section 2. (v) Here t_0 is still enabled but t_1 is not.
Fire $\tau_1 = t_0$ and obtain M_2'. By the rules in Section 2,
another transition $t_2 = accept$ is enabled. Set $\tau_g' =$
$generate \cdot fetch \cdot generate$, $M_g' = M_2'$, and $i = 2$. Go back to
step (4) of the strategy.
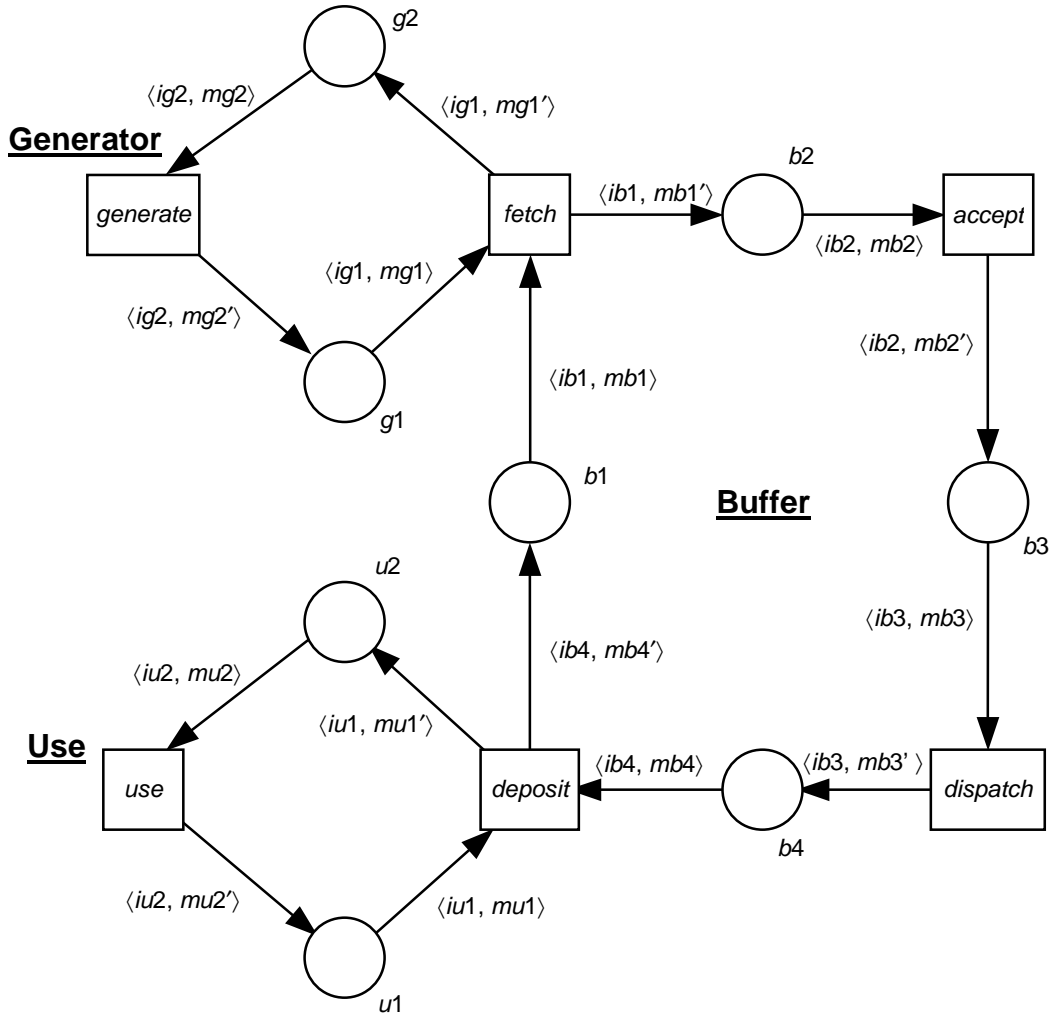
$ig1$, $ig2$, $iu1$, $iu2$, $ib1$, $ib2$, $ib3$, $ib4$: *ObjectName* in the form [*type*: *Type*, *id*: *Nat*], where *Type* denotes the set of object sorts, *Nat* denotes the set of natural numbers, *type* $\in$ {*g*, *u*, *b*}, *g* denotes generator, *u* denotes user, *b* denotes buffer, and *id* denotes object identity number.

$mg1$, $mg2$, $mg1'$, $mg2'$, *null*: *Message*. $mg1' = null$. $mg2' = generateMessage(ig2)$.

$mu1$, $mu2$, $mu1'$, $mu2'$, *nullmsg*: *FullMessage* in the form [*msg*: *Message*, *orig*: *Nat*], where *orig* denotes the originating object. $mu1' = mb4$. $mu2' = nullmsg$.

$mb1$, $mb2$, $mb3$, $mb4$, $mb1'$, $mb2'$, $mb3'$, $mb4'$: *FullMessage*. $mb1' = [mg1, num(ig1)]$. $mb2' = mb2$. $mb3' = mb3$. $mb4' = nullmsg$.

Figure 1. Case Study

Finally, we have obtained the ballpark goal sequence τ_g' = *generate · fetch · generate · accept · generate · fetch · dispatch · generate · fetch · accept · deposit · generate · fetch · accept · dispatch.* Obviously, in this case we have $ET(M_g') = T$, and hence the ballpark goal set TC_g' is $\{\tau_g' \cdot generate, \tau_g' \cdot fetch, \tau_g' \cdot accept, \tau_g' \cdot dispatch, \tau_g' \cdot deposit, \tau_g' \cdot use\}$. This can then be followed-up with class-level testing of observational equivalence [1].

We have implemented, in Java, a system consisting of these components. We have embedded 30 different mutants into the program. The above strategy in generating the test set TC_g' can kill all the mutants.

6. Conclusion

Based on an OBJSA net/CLOWN specification, we have proposed a strategy to select synchronization sequences for testing non-deterministic concurrent object-oriented software. We have reported an effectiveness case study and experiments on the proposed strategy. More case studies and experiments are being planned.

7. Acknowledgement

8. References

[1] H. Y. Chen, 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.

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

[3] D. C. Kung, 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.

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

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

[6] R. H. Carver 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.

[7] H. Zhu 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.

[8] H. Y. Chen. 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.

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

[10] E. Battiston, 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.

[11] H. Y. Chen, Y. X. Sun, and T. H. Tse, "A scheme for dynamic detection of concurrent execution of object-oriented software", in *Proceedings of the 2003 IEEE International Conference on Systems, Man, and Cybernetics* (*SMC 2003*), IEEE Computer Society Press, Los Alamitos, California, 2003.

[12] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, Boston, 2000.

[13] S. C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4): 334–377, 1996.

[14] H. S. Hong, S. D. Cha, D. H. Bae, and H. Ural. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability*, 10(4): 203–257, 2000.

[15] T. H. Kim, I. S. Hwang, M. S. Jang, and J. Y. Lee. Test case generation of a communication protocol by an adaptive state exploration. *Computer Communications*, 24(13): 1242–1255, 2001.

[16] D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul. Conformance testing of protocls specified as communicating finite state machines: a guided random walk based approach. *IEEE Transactions on Communications*, 44(5): 631–640, 1996.

[17] P. Tripathy and B. Sarikaya. Test generation from LOTOS specifications. *IEEE Transactions on Computers*, 40(4): 543–552, 1991.

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