<table>
<thead>
<tr>
<th>Title</th>
<th>A Strategy for Selecting Synchronization Sequences to Test Concurrent Object-Oriented Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Chen, HY; Sun, YX; Tse, TH</td>
</tr>
<tr>
<td>Citation</td>
<td>Proceedings - ieee Computer Society’s International Computer Software And Applications Conference, 2003, p. 198-203</td>
</tr>
<tr>
<td>Issued Date</td>
<td>2003</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10722/48445">http://hdl.handle.net/10722/48445</a></td>
</tr>
<tr>
<td>Rights</td>
<td>©2003 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.; This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.</td>
</tr>
</tbody>
</table>
A Strategy for Selecting Synchronization Sequences to Test Concurrent Object-Oriented Software

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  
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. They did not cover other concurrency properties. 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 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 superposed 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) \(P \cap T = \emptyset\)
(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 \(\nu = \{x \in V \mid (x, v) \in F\}\) is called its pre-set, and the set of arcs \(\nu^o = \{y \in V \mid (v, y) \in F\}\) is called its post-set.

(b) Extended SA Nets

Given a net \((P, T, F)\), an extended SA net \(N\) is a tuple \((P, T, F, W, \Pi)\) 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 \(OP \cap CP = \emptyset\).
(ii) \(OT\) is a set of open transitions and \(CT\) is a set of closed transitions such that \(OT \cup CT = T\) and \(OT \cap CT = \emptyset\).
(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) \(W: F \to \text{Nat}\) 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) \(\Pi\) is a partition of \(P\), dividing it into \(m\) subsets \((\Pi_i)_{i=1}^{m}\) such that \(\Pi_i \cup \Pi_j \cup \ldots \cup \Pi_m = P\) and \(\Pi_i \cap \Pi_j = \emptyset\) for \(i \neq j\). Each \(\Pi_i\) 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 \(\Pi_i\) to \(t\) is the same as the total weights of all the arcs from \(t\) to the places in \(\Pi_i\).

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 \(N = (P, T, F, W, \Pi)\) be an extended SA net and \(SPE = (S, \Sigma, E)\) be an algebraic specification, where \(S\) is a set of sorts (or object classes), \(\Sigma\) is a family of operation symbols, and \(E\) is a set of equational axioms, possibly conditional. An OBJSA component is a SPE-inscribed net \((N, \text{ins}, SPEC)\), where \(\text{ins} = (\varphi, \lambda, \eta)\) satisfies the following conditions:

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

(ii) \(\lambda: F \to \text{Terms}_{\Sigma}[X]\) 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 \(p \in t^o\), the label of the arc \(f = (p, t)\) is the concatenated string \(\lambda_X t\). Each \(\lambda_X t\) is a variable of sort \(\varphi(p)\). Let \(X\) be a list of variables that label the input arcs of \(t\), and \(\lambda_X t\) 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 \(\lambda_X t\). Each \(\lambda_X t\) is a term of the form \(\lambda_X t\) of sort \(\varphi(q)\) such that \(\lambda_X t\) is the same as \(\lambda_X t\) for some \(x_i = (n_i, d_i)\) in \(X\), and for some operation \(\sigma\), that specifies the change of the data content due to the transition \(t\).

(iii) \(\eta: T \times 2^X \to \text{Bool}\) is a guard function that assigns a pre-condition \(\eta(t)(X)\) 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 \(\varphi(p)\).

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 \( \beta_t: X_t \rightarrow \text{Terms}_{\mathbb{S},\mathbb{X}} \) that substitutes every variable \( x \in X_t \) by a ground term (that is, a term without variables) of sort \( \varphi(p) \) some \( p \in \mathcal{P} \).

For a place \( p \in P \), a transition \( t \in T \), and a firing mode \( \beta_t \), if \( p \in \mathcal{P}_t \), then \( IN(p, t) \) is defined as \( \{ \beta_t(x) \mid i = 1, 2, ..., W(p, t) \} \); otherwise \( IN(p, t) \) is defined as the empty set. Similarly, if \( p \in \mathcal{P}_t \), then \( OUT(t, p) \) is defined as \( \{ \beta_t(y(X_i)) \mid j = 1, 2, ..., t_i \} \); otherwise \( OUT(t, p) \) is defined as the empty set. Given a marking \( M \), a transition \( t \in T \), and a firing mode \( \beta_t \), if \( \eta(t)(\beta_t(x)) = \text{true} \) and \( IN(p, t) \subseteq M(p) \) for every place \( p \in \mathcal{P}_t \), then \( t \) is said to be \( \beta_t \)-enabled at \( M \). In this case, \( t \) may fire in mode \( \beta_t \). 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 \( \text{Osn} \), we say that a marking \( M \) is reachable if, starting from the initial marking \( M_0 \), we can obtain \( M \) by firing a sequence \( \tau \) of consecutively enabled transitions. In this case, we call \( \tau \) an enabled sequence and write \( M_0 \xrightarrow{\tau} M \). If \( \tau = \text{null} \), \( M = M_0 \).

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

\( \text{md}(\text{Osn}) \) must exist, because \( ed(\text{Osn}, M) \) for any \( M \in M^* \) must be a non-negative integer satisfying \( ed(\text{Osn}, M) \leq \lvert T \rvert \). The goal marking \( M_0 \) corresponding to \( \text{md}(\text{Osn}) \), however, is not necessarily unique. At \( M_0 \), the firings of transitions in \( \text{Osn} \) have the maximum non-determinacy and 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 \( M_0 \) in our testing. This is the motivation and the general idea of our strategy for selecting synchronization sequences.

Suppose \( M_0 \xrightarrow{\tau} M_1 \). Let \( TC^g = \{ \tau \cdot t_i \mid t_i \in ET(M_2^g) \} \), where \( \tau \cdot t_i \) denotes the firing of \( t_i \) immediately after \( \tau \). We say that \( \tau \) 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. If the execution of all the elements of \( TC \) causes each transition in \( T \) to fire at least once, we say that \( TC \) is a transition-covering set. For any given transition \( t \in T \), if there is another enabled transition \( t_i \) at \( M_0 \), the problem is difficult because \( M^* \) is infinite in general. Furthermore, the construction of a transition-covering set is also generally difficult.

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

Suppose \( t_0 \) is the GIST of \( \text{Osn} \) with \( M_g \). Our heuristic strategy to seek a ballpark goal marking \( M_g^b \) and the corresponding ballpark goal sequence \( \tau_g^b \) includes the following procedure:

1. set \( \tau_g^b = \text{null} \) and \( \tau_g^b = M_g \);
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 \( \tau_g^b \) and \( \tau_g^b \), and exit;
   - if \( t_0 \) is still enabled, {
     - if there is no other enabled transition \( t_1 \), return \( \tau_g^b \) and \( \tau_g^b \), and exit;
     - if there is another enabled transition \( t_1 \), {
       - set \( \tau_g^b = \tau_g^b \cdot t_0 \), \( M_g^b = M_1 \), \( M_2^g = M_1 \), and \( i = 1 \);
     }
   };

if there are adequately many enabled transitions at \( M'_g \), then return \( \tau'_g \) and \( M'_g \), and exit; otherwise
\[
M' \xrightarrow{t_i} M_{i+1};
\]
(5) at \( M_{i+1} \):
if \( t_0 \) is not enabled, return \( \tau'_g \) 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
\( \tau'_g \) and \( M'_g \), and exit;
if there is another enabled transition \( t_{i+1} \), {
set \( \tau'_g = \tau'_g \cdot t_i \cdot \tau_g, M'_g = M_{i+1}' \), 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 \( t_0, t_1, \ldots, t_i \) several times can enable all of \( t_0, t_1, \ldots, t_i \), {
let \( \tau_g \) be the corresponding sequence of the fired transitions and let \( M_{i+1}' \) be the marking obtained, that is, \( M_{i+1} \xrightarrow{\tau_g} M_{i+1}' \);
if there is no other enabled transition \( t_{i+1} \) at \( M_{i+1}' \), return \( \tau'_g \) and \( M'_g \), and exit;
if there is another enabled transition \( t_{i+1} \) at \( M_{i+1}' \), {
set \( \tau'_g = \tau'_g \cdot t_i \cdot \tau_g, M'_g = M_{i+1}' \), and \( i = i+1 \);
go to (4);
}
else return \( \tau'_g \) 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 \( \tau'_g \) and the corresponding \( M'_g \), we can construct the ballpark goal set \( TC'_g = \{ \tau_g \cdot t_i | t_i \in ET(M'_g) \} \) and take \( TC'_g \) as a set of test cases.

Since a transition in OBJSA net represents an operation in the implementation, each test case in \( TC'_g \) represents a sequence of operations in the implementation. We call it a synchronization sequence. Because of the non-determinacy of concurrent programs, we must use a replay technique to execute each synchronization sequence in \( TC'_g \). Details of replay techniques can be found in [13].

Note that \( TC'_g \) may not cover all the transitions of \( Osn \). In this case, and if \( TC'_g \) cannot reveal errors, we need also to construct other sets of test cases covering \( T \setminus (ET(M'_g) \cup \{ t \} \}) \) as supplements.

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 asynchronously exchanging messages through four buffers. These three constituents can be specified by OBJSA open components Generator, User, and Buffer, respectively. Each component is further identified by a natural number. The OBJSA net specifying the integrated system is shown in Figure 1. Its initial marking \( M_0 \) is as follows:

\[
M_0(g2) = \{(g, 1), null\}, \{(g, 2), null\}, \{(g, 3), null\}, \{(g, 4), null\};
M_0(u2) = \{(u, 1), nullmsg\}, \{(u, 2), nullmsg\}, \{(u, 3), nullmsg\}, \{(u, 4), nullmsg\}, \{(u, 5), nullmsg\};
M_0(b1) = \{(b, 1), nullmsg\}, \{(b, 2), nullmsg\}, \{(b, 3), nullmsg\}, \{(b, 4), nullmsg\};
M_0(p) = \emptyset \text{ for } p \in P \setminus \{g2, u2, b1\}.
\]

Following the definition in Section 4, we have obtained the GIST \( t_0 = \text{generate} \). Following the proposed strategy, we have the following process: (i) Set \( \tau'_g = \text{null} \) and \( M'_g = M_g \). (ii) Fire \( t_0 \) and obtain \( M_g \) by the rules in Section 2. (iii) Here \( t_0 \) is still enabled. By the definition in Section 2, another transition \( t_1 = \text{fetch} \) is enabled. Set \( \tau'_g = \text{generate} \), \( M'_g = M_g \), \( M'_g = M_g \), and \( i = 1 \). (iv) Fire \( t_1 = \text{fetch} \) and obtain \( M_g \) by the rules in Section 2. (v) Here \( t_0 \) is still enabled but \( t_1 \) is not. Fire \( t_1 = \text{fetch} \) and obtain \( M'_g \). By the rules in Section 2, another transition \( t_2 = \text{accept} \) is enabled. Set \( \tau'_g = \text{generate} \cdot \text{fetch} \cdot \text{generate} \cdot \text{accept} \cdot \text{generate} \cdot \text{fetch} \cdot \text{dispatch} \cdot \text{generate} \cdot \text{fetch} \cdot \text{accept} \cdot \text{generate} \cdot \text{fetch} \cdot \text{accept} \cdot \text{dispatch} \). Obviously, in this case we have \( ET(M'_g) = T \) and hence the ballpark goal set \( TC'_g \) is \( \{ \tau'_g \cdot \text{generate}, \tau'_g \cdot \text{fetch}, \tau'_g \cdot \text{accept}, \tau'_g \cdot \text{dispatch}, \tau'_g \cdot \text{deposit}, \tau'_g \cdot \text{use} \} \).

We have implemented, in Java, a system consisting of these components. We have embedded 30 different mutants into the program. The above \( TC'_g \) can kill all the mutants.

---

3 That is, in the most recent \( m \) loops, the number of enabled transitions at \( M'_g \) does not increase, where \( m \) is a parameter specified by the user.
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 ∈ {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
6. Conclusion

Based on an OBJSA net specification, we have proposed a strategy to select synchronization sequences for testing 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

We are grateful to Shuang Quan Li of Jinan University for implementing the system and conducting the experiments and tests in the case study.

8. References


