<table>
<thead>
<tr>
<th>Title</th>
<th>A scheme for dynamic detection of concurrent execution of object-oriented software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Chen, HY; Sun, YX; Tse, TH</td>
</tr>
<tr>
<td>Issued Date</td>
<td>2003</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10722/48444">http://hdl.handle.net/10722/48444</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 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.

0-7803-7952-7/03/$17.00 @2003 IEEE 4828
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 \( v \in V \), \( v^o = \{ y \mid y \in V \land (y, v) \in F \} \) is called the pre-set of \( v \), and \( v^p = \{ y \mid y \in V \land (v, y) \in F \} \) 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. \( W : F \rightarrow \text{Nat} \) is the arc weight function, where \( \text{Nat} \) denotes the set of natural numbers. In particular, \( W(f) = 1 \) for every open arc \( f \). \( \Pi \) is a partition of \( P \) into disjoint classes \( \Pi_1, \Pi_2, \ldots, \Pi_n \) such that every \( \Pi_i \) contains either open places only or closed places only, and for every \( t \in T \), \( \Sigma_{p \in \Pi}(v^o)W(p, t) = \Sigma_{p \in \Pi}(v^p)W(t, p) \).

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 \( SPEC = (S, \text{Opt}, Eq) \), an OBJSA component is a SPEC-inscribed net \( (N, ins, SPEC) \) with an initial marking (or initial state) \( M_0 \), where \( ins = (\phi, \lambda, \eta) \) is a SPEC-inscription of \( N \) such that:

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

(b) \( \lambda : T \rightarrow \Sigma_\phi \) is a \( \phi \)-respecting arc labeling function, which assigns labels to the arcs surrounding every transition as follows: For every \( t \in T \), let \( \lambda(t) = \{ \lambda_1(t), \ldots, \lambda_p(t) \} \) and \( t^o = \{ q_1, q_2, \ldots, q_b \} \). For every arc \( f = (p_i, t) \), if \( f \) is open, its label is a variable \( x_{i1} \) of sort \( \phi(p_i) \); otherwise its label is of the form \( x_{i1} \leftrightarrow x_{i2} \leftrightarrow \ldots \leftrightarrow x_{ij} \), where each \( x_{ij} \) is a variable of sort \( \phi(p_i) \). Let \( X_t \) be a list of variables that label the input arcs of \( t \). For every arc \( f = (t, q_j) \), if \( t \) is open, its label is a term \( y_{j1}(X_t) \); otherwise its label is of the form \( y_{j1}(X_t) \leftrightarrow y_{j2}(X_t) \leftrightarrow \ldots \leftrightarrow y_{j1}(X_t) \), where each \( y_{j1}(X_t) \) is a term of sort \( \phi(q_j) \). Furthermore, for each variable \( x_{ij} \), \( x_{ij} = <n_{ij}, d_{ij}> \) in \( X_t \), there exists a unique term \( y_{j1}(X_t) = <n_{ij}, d_{ij}> \) of sort \( \phi(q_j) \) such that \( n_{ij} = n_{ij} \) and \( d_{ij} = \sigma(...) \) for some function \( \sigma \) that specifies the change of the data content due to the transition \( t \).

(c) \( \eta : T \rightarrow \text{Bool} \) 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 \( \phi(p) \), 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 \( \phi(op) \).

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].
Producers

Consumers

c2
--

Agents

npl, np2, ncl, nc2, nal, na2, na3, na4: ObjectName in the form \([\text{type}: \text{Type}, \text{id}: \text{Nat}]\), where Type denotes the set of object sorts, Nat denotes the set of natural numbers, \(\text{type} \in \{p, c\}\), \(p\) denotes producer, \(c\) denotes consumer, and \(\text{id}\) denotes the object identifier.

dal, da2, da3, da4, dal*, da2*, da3*, da4*: FullMessage in the form \([\text{msg}: \text{Message}, \text{dest}: \text{Nat}]\), where dest denotes a destination object. \(da1* = dp1\). \(da2* = nullmsg\). \(da3* = da2\). \(da4* = nullmsg\).

dp1, dp2, dp1*, dp2*, nullmsg: FullMessage. \(dp1* = nullmsg\). \(dp2* = \text{produceMessage}(np2)\).

dc1, dc2, dc1*, dc2*, null: Message. \(dc1* = \text{msg}(da4)\). \(dc2* = null\).

pr1, pr2, pr3: Bool. \(pr1 = (\text{type}(na1) = p) \land (\text{id}(na1) = \text{id}(np1))\).
\(pr2 = (\text{type}(na2) = p) \land (\text{type}(na3) = c) \land \text{dest}(da2) = \text{id}(na3))\).
\(pr3 = (\text{type}(na4) = c) \land (\text{id}(na4) = \text{id}(na1))\).

Figure 1. OBJSA Net Specifying the System in Example 1
Given an OBJSA component with a transition \( t \in T \) such that \( t = b_1, p_2, \ldots, p_n \) and \( t' = \{ q_1, q_2, \ldots, q_b \} \), a firing mode for \( t \) is an assignment function \( \beta : X_t \to T_{\text{set}} \) that associates a ground term of sort \( \varphi(p_i) \) to every variable \( x_{ij} \) in the list \( X_t \).

Given a firing mode \( \beta_n \) for each place \( p \in P \), \( \text{IN}(p, t) \) is defined as follows: (a) \( \text{IN}(p, t) = \{ \beta_i(x_{ij}) \} \) if \( p = p_i \in t \cap CP \); (b) \( \text{IN}(p, t) = \{ \beta_i(x_{ij}) \} \) if \( p = p_i \in t \cap OP \); (c) \( \text{IN}(p, t) = \emptyset \) otherwise. Given a marking \( M \), a transition \( t \in T \) and a firing mode \( \beta_n \), we say that \( t \) is \( \beta_n \)-enabled at \( M \) if, for each place \( p_i \in t \), \( \text{IN}(p_i, t) \subseteq M(p_i) \) and \( \eta(t, \beta_n) = \text{true} \).

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:

\[
M_0(p_2) = \{ [p, 1], \text{nullmsg}, [p, 2], \text{nullmsg}, [p, 3], \text{nullmsg}, [p, 4], \text{nullmsg}, [p, 5], \text{nullmsg} \};
\]

\[
M_0(c_2) = \{ [c, 1], \text{null}, [c, 2], \text{null}, [c, 3], \text{null}, [c, 4], \text{null} \};
\]

\[
M_0(a_1) = \{ [p, 1], \text{nullmsg}, [p, 2], \text{nullmsg}, [p, 3], \text{nullmsg}, [p, 4], \text{nullmsg}, [p, 5], \text{nullmsg}, [c, 1], \text{null}, [c, 2], \text{null}, [c, 3], \text{null}, [c, 4], \text{null} \};
\]

\[
M_0(p) = \emptyset \text{ for } p \in P - \{ p_2, c_2, a_1 \}.
\]

**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 \( t_1 \) and \( t_2 \) (or their corresponding operations) can be fired simultaneously at \( M_0 \), we say that \( t_1 \) and \( t_2 \) are a concurrent pair at \( M_i \). In formal terms, \( t_1 \) and \( t_2 \) are a concurrent pair at \( M_i \) if and only if \( t_1 \) and \( t_2 \) are enabled respectively and \(-3p (p \in t_1 \cap t_2 \land \forall \beta \text{IN}(p, t_1) \cup M(p) - \text{IN}(p, t_1)) \). For simplicity, let \( I_1, I_2, \) and \( M \) denote \( \text{IN}(p, t_1), \text{IN}(p, t_2), \) and \( M(p) \), respectively. It can easily be proved that if \( I_1 \subseteq M \) and \( I_2 \subseteq M \), then \( t_1 \subseteq M - I_1 \) implies \( I_1 \cap I_2 \neq \emptyset \) and \( I_2 \subseteq M - I_1 \) if and only if \( I_1 \subseteq M - I_2 \).

Hence, the expression \( \text{IN}(p, t_2) \cup M(p) - \text{IN}(p, t_1) \) above can be replaced by \( \text{IN}(p, t_1) \cup M(p) - \text{IN}(p, t_2) \). In other words, \( t_1 \) and \( t_2 \) will be a concurrent pair at \( M_i \) if and only if \( t_1 \) and \( t_2 \) are enabled respectively and \(-3p (p \in t_1 \cap t_2 \land \forall \beta \text{IN}(p, t_1) \cup M(p) - \text{IN}(p, t_2)) \).

Let \( \tau \) be a sequence of individual or concurrent transitions. The notation \( M_i = \tau \rightarrow M_f \) means that, starting with the marking \( M_i \), we can consecutively fire the transitions in \( \tau \) to obtain the marking \( M_f \). When \( \tau \) is null, \( M_i = M_f \). The notation \( M_i = \tau \rightarrow M_f \) means that simultaneously firing \( t_1 \) and \( t_2 \) at \( M_i \) will obtain the marking \( M_f \). In fact, \( M_i = \tau \rightarrow M_f \) can be taken as a test case.

If \( M_0 = \tau \rightarrow M_0 \), where \( M_0 \) is the initial marking of \( Osn \), we say that \( M_0 \) can be reached by \( \tau \). Thus, the test case \( M_i = \tau \rightarrow M_i \) can be written as \( \tau \cdot t_1 \parallel t_2 \), which means that we can reach \( M_i \) if we start from \( M_0 \) consecutively fire the individual or concurrent transitions in \( \tau \), and then simultaneously fire the concurrent pair \( t_1 \) and \( t_2 \).

Let \( t_0 \) be the greatest source transition of a given OBJSA net \( Osn \). Our scheme for selecting concurrent test cases of the form \( M_i = \tau \rightarrow M_i+1 \), or \( \tau \cdot t_1 \parallel t_2 \), contains the following steps:
(1) set $M_c := M_0$, $T_c := \{t_0\}$, $\tau_c := \text{null}$, 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 $M_n$ after firing $t_0 \cdot t_1 \cdot \ldots \cdot t_k$, that is, $M_c ^{t_0 \cdot t_1 \cdot \ldots \cdot t_k} = M_n$, and if we can find $t_{k+1}$ ($\not\in T_c$) and $t_k (\in T_c)$ that can be fired simultaneously at $M_n$,

then {
    
    $M_i ^{t_k} \rightarrow M_{i+1}$, $T_c := T_c \cup \{t_{k+1}\}$, $	au_i := \tau_c \cdot t_0 \cdot t_1 \cdot \ldots \cdot t_k$, and $\tau_c := \tau_i \cdot t_k$; 
    
    return a test case $\tau_i \cdot t_{k+1} || t_{k+2}$;

} 
else exit from the scheme;

(3) $i := i + 1$;
if we can find $t_{k+1}$ ($\not\in T_c$) and $t_{k+2}$ ($\not\in T_c - \{t_0\}$) that can be fired simultaneously at $M_n$,

then {

$M_i ^{t_{k+1}} \rightarrow M_{i+1}$, $T_c := T_c \cup \{t_{k+2}\}$, $	au_i := \tau_c$, and $\tau_c := \tau_i \cdot t_{k+1}$; 

return a test case $\tau_i \cdot t_{k+2}$; 

goto (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 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 $\tau_i \cdot t_{k+1} || t_{k+2}$ 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:

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

where $t_0 = \text{generate}$, $t_1 = \text{send}$, $t_2 = \text{exchange}$, $t_3 = \text{receive}$, and $t_4 = \text{consume}$.

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


