JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 20, 57-83 (2004)

Mining Control Patterns from Java Program Corpora*

DENG-JYI CHEN, CHUNG-CHIEN HWANG, SHIH-KUN HUANG+
AND DAVID T. K. CHEN++
Department of Computer Science and Information Engineering
National Chiao Tung University
Hsinchu, 300 Taiwan
+*Institute of Information Science*
Academia Sinica
Taipei, 115 Taiwan
++*Department of Computer and Information Science*
Fordham University
Bronx, N.Y., U.S.A.

Java programming, based on the Object-Oriented (OO) paradigm, has played a major role in program design and implementation due to the fact that it is extensible, maintainable, and reusable in software system construction. Experiences with using Java programming have indicated that there also exist disadvantages with respect to its execution inefficiency and complicated runtime behaviors. Code-patterns are statically recurring structures specifically related to a programming language. They can be used in parallel to help programmer design software systems to solve particular problems. In opposition to the role of code-patterns in assisting compilation, control-patterns are dynamically recurring structures invoked during program execution time. They can be used to understand the run-time behaviors of OO-programs with respect to the underlying architecture, such as Java-VM. A control pattern describes the model of control transfer among objects during OO program execution. In this paper, several control patterns are proposed and discussed. We have analyzed and collected several control patterns from several Java program corpora. The experimental results show that control patterns do exist and provide information for quantitative analysis. Simple patterns, compound patterns, and complex patterns have different ratios depending on their source programs. Collected control patterns can be used to provide guidelines for Java programmers so that they can write more effective Java programs.

Keywords: OOP, control patterns, data mining, java VM, code patterns, benchmark design, program optimization, static and dynamic analysis

1. INTRODUCTION

Over the years, issues related to performance improvement in the run-time environments of OO systems have been studied and researched [1, 2, 6, 9, 12]. Since run time behaviors with respect to various application domains and code patterns differ substantially, optimization should be domain or pattern specific [1]. Programs designed and

Received January 31, 2003; accepted July 4, 2003.
Communicated by Ming-Syan Chen.
* This research work is supported by the National Science Council in Taiwan under the contract number 88-2213-E-009-007 and an earlier version of this paper has been presented at the ICS 2002.

written based on an object-oriented approach are more extensible, maintainable, and reusable than those produced based on traditional procedural-oriented approach [3, 4, 14, 15]. However, complex runtime behaviors created due to the message sending and polymorphism mechanisms have become the disadvantage associated with object-oriented programming. The traditional concept of program analysis is not broad enough to represent the real runtime behavior of OO programs. For traditional procedural-oriented languages, the control flows can be divided into three types: sequential execution, conditional branch, and unconditional branch. In pure object-oriented languages, such as Smalltalk, there are only two types of control flows: sequential execution and message sending flows. Sequential execution flow is not different from procedure-oriented programs. The instructions are executed one by one without jumping to other places. Message sending means that execution switches to other groups of instructions. During object-oriented program execution, the action of invoking a method to be executed is known as control transfer. A method invocation sequence records the entire control transfer, which takes place during program execution. There might be some recurrence patterns in this method invocation sequence, and these recurrence patterns of control transfer are called control patterns. These patterns typically represent the run-time behavior of object-oriented programming. In other words, the more precisely the patterns are found, the more we can explore the real runtime world. With this in mind, our goal is to obtain the runtime method invocation sequences of object-oriented programs first, and then to build tools to analyze the invocation sequences to better understand the properties of runtime behavior. Then, control patterns can be found by analyzing patterns. Thus, the performance measurement and performance improvement based on the control patterns can be measured hereafter.

In this research, Java was chosen as our experimental language due to its popularity in the OO community and its flexibility under different platforms. Java is an object-oriented programming language developed by Sun Microsystems. Java program will be executed on a virtual machine, called the Java Virtual Machine (JVM). Java programs are first compiled into byte codes, which are then executed on the JVM [5]. If run-time information about Java program is required, only the JVM needs to be modified. There is no need to recompile the programs. The objectives of this research were to acquire run-time method invocation sequences during object-oriented program execution, and to build a tool to analyze invocation sequences. A runtime model based on behaviors among objects will be proposed. It helps us to understand the runtime behaviors of the program. A control pattern-mining tool has been designed to explore runtime behavior and to quantify the measured results. Specifically, we have analyzed and collected several control patterns from several Java program corpora. The experimental results show that control patterns do exist. Simple patterns, compound patterns, and complex patterns have different ratios respectively depending on the source programs. The collected control patterns can be used to provide guidelines for Java programmers to write more effective Java programs.

2. CONTROL PATTERNS

The progress of an object-oriented program is regarded as the lifetime of objects during program execution. During the execution of an object-oriented program, the ac-

tion of invoking a method to be executed is known as control transfer. A method invocation sequence keeps track of all the control transfers that occur during program execution. Although the real action of method invocation is control transfer among objects, in terms of the behavior that occur during program execution, we can transform control transfer into a different kind of transfer between receiver classes. The call graph of a program represents the possible callees at each call site in each procedure. Inter-procedural analyses typically produce summary results showing the effect of callers at each call site as well as summaries of the effect of callers at each procedure entry. Unfortunately, in the presence of dynamically dispatched messages or the invocation of computed functions, the set of possible callees at each call site is difficult to predict precisely. The reason is that different input data will result in different execution paths. The calling relationship among objects analyzed based on the method invocation sequence is more concrete than call graph.

A control pattern includes a directed graph that is a small subgraph of a call graph plus two functions: the constraint output function and constraint Boolean function. It can explain which subgraph is really executed in the call graph as well as the quantitative results of this subgraph. Moreover, the constraint output function and constraint Boolean function describe the real execution path in the subgraph. In addition, class hierarchy analysis [7-9] exploits information about the structure of the class inheritance graph. An execution log pattern identifies the movement of control in a class inheritance graph. It is possible that the frequency of dynamic message binding to be reduced by splitting and combining of classes. A method invocation sequence records all the control transfers that occur during program execution. While a program segment running with specific input data, a control pattern can be extracted through a method invocation sequence.

2.1 Semantic Meanings of Control Patterns

We have evaluated several kinds of simple control patterns (consecutive patterns, loop-N patterns [12], sequence patterns, dispatch patterns, and join patterns [13]) according to language features. A formal definition of control patterns has been discussed in [2, 13]. In general, control patterns are divided into three groups: simple control patterns, compound control patterns, and complex control patterns. Control patterns created by language features are classified as simple control patterns. Compound control patterns are combinations of simple control patterns. Control patterns that combine with simple and compound patterns are defined as complex control patterns. Like a sequential statement that is executed sequentially, some program segments in OO programs may be executed in the same manner. We call them sequence patterns. The semantic meanings of these three groups of control patterns are introduced in the following.

2.1.1 The semantic meanings of simple control patterns

Take the segment of the program shown in Fig. 1 as an example. It traverses a tree and gives each node a number to represent its traversal order. Regardless of how many classes and methods are shown in the whole program, during the execution of this program segment, only 2 classes and 7 methods are involved. They are *stack* and *ce* classes, and *stack.empty()*, *stack.push()*, *stack.pop()*, *ce.setOK()*, *ce.setValue()*, *ce.hasoreChildren()* and *ce.nextChild()* methods.

```
while (!stack.empty()) {
   ce = stack.pop();
   if (ce.traverse()) {
      ce.setOK();
   }
   else {
         ce.setValue (value++);
         stack.push (ce);
         while (ce.hasMoreChildren())
               stack.push (ce.nextChild());
   }
}
```

Fig. 1. Segment of a tree traversal program.

A method invocation consists of three parts: receiver class, method class, and method. Fig. 2 shows the method invocation sequence produced by running the program segment shown in Fig. 1. Consider the method invocation sequence shown in Fig. 2. The 1st and the 2nd method invocations are instances of consecutive patterns because they are consecutive and their receiver classes are the same (stack). "CP" is the abbreviation of "Consecutive Patterns". Consider the method invocation beginning at 6 and ending at 14, the same method invocation is repeated for every three-method invocations. As a result, they are treated as an instance of a Loop-3 pattern, abbreviated as "LP3." Directed graphs are used to illustrate the concept of LP3 in Fig. 3.

method invocation sequence

consecutive (1–2)

1 (stack)stack.empty()
2 (stack)stack.pop()
3 (ce)ce.traverse()
4 (ce)ce.setValue()
5 (stack)stack.push()
6 (ce)ce.hasMoreChildren()
7 (ce)ce.nextChild()
8 (stack)stack.push()
9 (ce)ce.hasMoreChildren()
10 (ce)ce.nextChild()
11 (stack)stack.push()
12 (ce)ce.hasMoreChildren()
13 (ce)ce.nextChild()
14 (stack)stack.push()
15 (stack)stack.empty()
16 (stack)stack.pop()
17 (ce)ce.traverse()
18 (ce)ce.setValue()
⋮

Loop-N (6–14)

Fig. 2. Method invocation sequence of the program segment shown in Fig. 1.

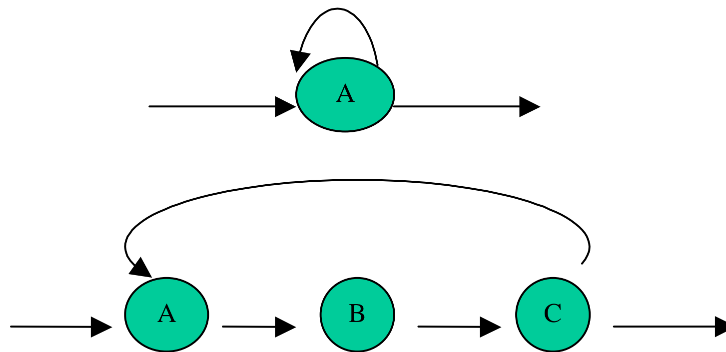

Fig. 3. Consecutive pattern and loop-3 pattern.

Let us look at another example shown in Fig. 4. In the left part of this figure is shown the definition of classes, whereas the middle part shows a program segment. Also, the right part shows the corresponding method invocation sequence. Considering the 1st to the 4th method invocations, they produce two execution log patterns (AB, AC). As a result, they form an instance of a dispatch pattern (DP) as shown in Fig. 5. The 5th to the 8th method invocations (BD, CD) form an instance of a join pattern (JP) as shown in Fig. 6.

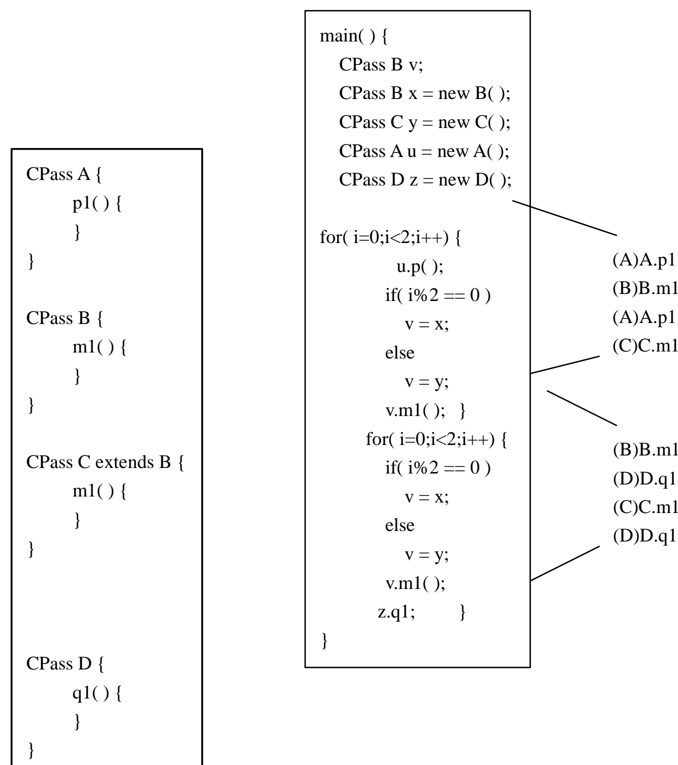

Fig. 4. Dynamic message sending examples.

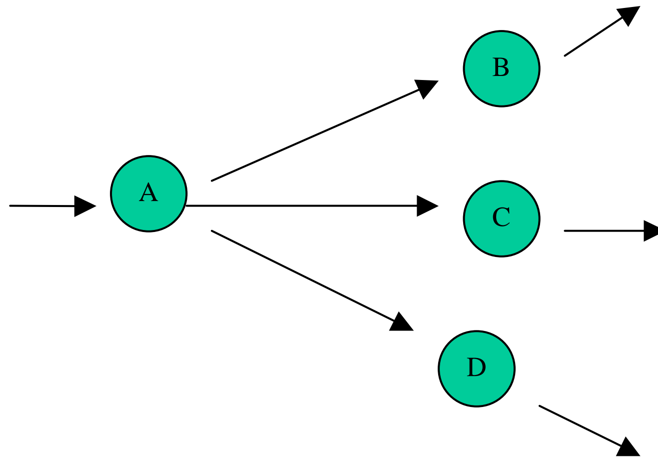

Fig. 5. Dispatch control pattern.

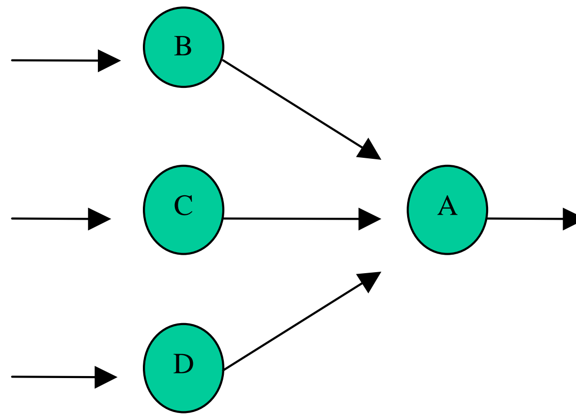

Fig. 6. Join control pattern.

2.1.2 Compound control patterns

A compound control pattern with G(E, V) is isomorphic to a simple control pattern with G'(E', V'). For each vertex of V', there can be either a compound control pattern or a simple control pattern. A compound control pattern has an architecture similar to that of a simple control pattern. The vertex of a simple control pattern can be either a simple control pattern or a compound control pattern. Fig. 7 shows that the main skeleton of a control pattern consists of a sequence pattern with 2 elements, and that the second element can either be a dispatch pattern, a consecutive pattern, a join pattern, or a compound control pattern.

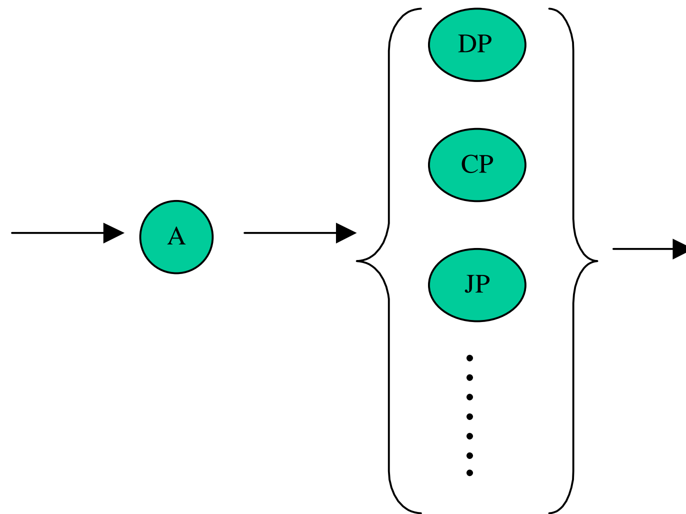

Fig. 7. Example of a compound control pattern.

2.1.3 Complex control patterns

In general, a directed graph is constructed for a set of execution log patterns. Every log pattern exists in the graph. Nevertheless, the directed graph does not belong to either a simple control pattern or a compound control pattern. The set of execution log patterns {ACDE, ABE, ACE} shown in Fig. 8 can be used as an example.

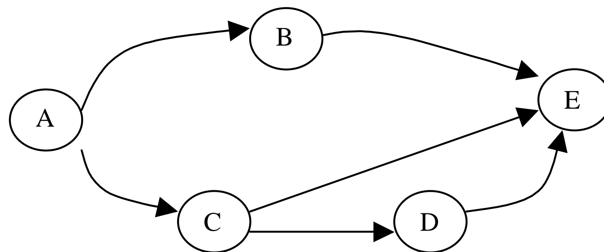

Fig. 8. Example of a complex control pattern.

In fact, a control pattern with G(E, V) is constructed based on a set S of execution log patterns. It is possible for a path from a source to a sink to be in G but not in S. Consider the set of execution log patterns {AB_1CD_1E, AB_2CD_2E} shown in Fig. 9; the path AB_1CD_2E is in the directed graph, and this is not the same execution log pattern as the one constructed in the set. Therefore, we design a constraint output function corresponding to each vertex and a constraint Boolean function corresponding to every edge. In the following section, a detailed definition of control patterns will be given.

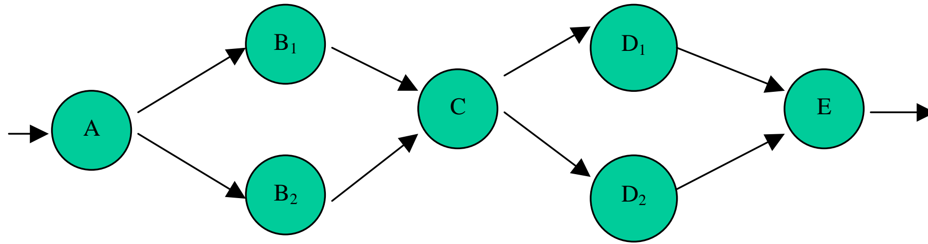

Fig. 9. Example of a complex control pattern.

2.2 Control Pattern Mining

When Java programs are executed on a modified JVM, a large database of events is given, where each event consists of a receiver class, method class, method, event order, and the items brought into the event. All the events can together be viewed as a sequence, where each event corresponds to a set of items. The list of events, labeled according to increasing event order, corresponds to a sequence. In this section, an algorithm is proposed to solve the problem of mining sequential patterns in such a database. Data mining is an application-dependent issue. Different applications may require different mining techniques. Method invocation results can be viewed as a sequence, where each method invocation or event can correspond to a set of items, a receiver class, method class, method, and event order. Each event order is a unique one. Mining the patterns of an execution log is the problem of finding a sequence of patterns related to an ordered list of method invocation. Mining of sequence patterns can be accomplishing using many algorithms, but the results are the same since the association rule, data classification, and data clustering was given to produce these control patterns. To be specific, we shall demonstrate some recurring patterns in the method invocation sequence. These recurring patterns of control transfer are called execution log patterns.

Clustering analysis [10, 11] helps us construct meaningful partitions of a large set of objects based on the methodology "divide and conquer" which decomposes a large scale system into smaller components to simplify design and implementation. There should be small distances between the elements of the same cluster and large distances between the elements of different clusters. A data mining algorithm to construct the control pattern corresponding to a set of execution log patterns has been presented in [13] and is recalled here. One can divide the problem of control pattern mining into two parts. The first part is the graph construction, and the second part is the constraint condition mining.

Part 1: Graph Construction

For a given set of *m* execution log patterns of the same control pattern, we will construct a directed graph.

Initially, we set up the activity control pattern table.

Algorithm setup-activity-control-pattern-table:

$\forall$ a, b$\in$ V, f(a, b) = f(b, a) = true

E.g:

Let V = {a, b, c}

a	a	true
a	b	true
a	c	true
b	a	true
b	b	true
b	c	true
c	a	true
c	b	true
c	c	true

Algorithm 1. /* Suppose that we have m execution log patterns, and that t_i is the length of the ith execution log pattern.

1. For i = 1 to m
2. For j = 2 to t_i
3. Set-activity-control pattern-table and let E = Ø.
4. do /* Assume that the ith execution log pattern is $u_1, u_2, \ldots, u_{ti}$
   if ($f(u_{j-1}, u_j)$) then $E = E \cup (u_{j-1}, u_j)$;
   $f(u_{j-1}, u_j)$ = false;

The time complexity of algorithm 1 is $O(\Sigma\, t_i)$, where i = 1 to m. Let t be the average length of $\{t_1, \ldots, t_m\}$; then the complexity is O(tm).

Part 2: Constraint Condition Mining

Given a set of m execution log patterns of the same control pattern and its corresponding directed graph G = (V, E), find the constraint output function o(v), $v \in V$, and the constraint Boolean function $f_{(u,\, v)}$ $(u, v) \in E$.

Algorithm breadth-search (G):

Let id(v) and od(v) be the incoming degree of v and the outgoing degree of v, respectively.

Algorithm 2.

1. $\forall\, v \in V$, $o(v) = \{\lambda\}$
2. While (v = breadth-search (G)) $\neq \lambda$ do
3. {
4. $O = \{\lambda\}$
5. if id(v) > 1 then $O = \{t \mid t \mid v \mid \alpha$ is an execution log pattern$\}$
6. if od(v) > 1 then o(v) = O
7. if $o(v) = \{\lambda\}$ then $f_{(v,v\text{-next})} = \{1\}$ and $f_{(v,\text{other vertices})} = \{0\}$
   else $f_{(v,v\text{-next})}(o(v)) = \{1\}$ and $f_{(v,\text{other vertices})} = \{0\}$
8. }

2.3 Mining Execution Log Patterns

In this section, we will describe how execution log patterns are looked up in a method invocation sequence. CP and LP appear frequently, so we split the problem of mining sequence pattern into two phases as described in the following.

1. CP Phase: Find the simple execution log patterns (Consecutive Patterns and Loop-N Pattern) and replace them with a control pattern identifier:

ACDCCCCCCCCCEFHIACABCABCABCABCABCKJEFDK
⇒ ACDCPEFHIALP3KJEFDK.

The modified method invocation sequence is the input of the sequence phase. Different simple control patterns are found separately. Only one control pattern is calculated for each pass of the method invocation sequence. Fig. 10 [6] illustrates the method that our analyzer uses to look up control patterns in a method invocation sequence. The method invocation sequence is fed into the predictor. The predictor maintains several internal states and uses them to predict the next method invocation. The output of the predictor is compared with the next method invocation input from the method invocation sequence. If the output of the predictor is the same as the input method invocation, then the input method invocation together with the method invocations in the predictor comprises an instance of the evaluated control pattern. The input method invocation is then fed into the predictor to update the internal states of the predictor.

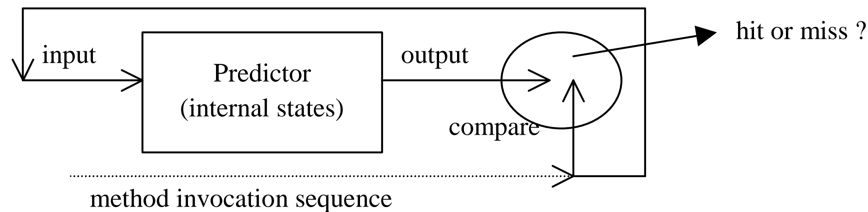

Fig. 10. Predictor for evaluating control patterns.

The predictor is configurable. The predictor is set up with different internal state configurations corresponding to different control pattern analyses. The predictor is set up with one internal state to record the previous method invocation and to evaluate the consecutive pattern. The internal state is used as output for comparison with next input method invocation. The predictor is set up with three internal states to record the previous three method invocations and to evaluate the loop-3 pattern. The third internal state is used as output for comparison with the next input method invocation. When these techniques are used, the consecutive pattern and loop-N pattern in a method invocation sequence can be easily found.

2. Sequence Phase: Multiple passes over the modified method invocation sequence are conducted. In each pass, we start with a seed set of large sequences and call a sequence satisfying a minimal support constraint for a large sequence. In the first pass, the seed set contains only 1-sequences with minimum support. The details of the algorithms are given in the following:

1. L_1 = {large 1-itemsets}
2. For (k = 2; $L_{k-1} \neq \emptyset$; k++)
3. $O_{k-1} = L_{k-1}$
4. C_k = cc-gen(L_k-1) // New candidates
5. forall candidate $c \in C_k$ do begin
6. forall ($\forall p \subset L$, p = c_{k-1} and the next token of p = c.item$_k$) do
7. c.count++;
8. end
9. $L_k = \{c \in C_k \mid c.count \geq minsup\}$
10. $O_{k-1} = O_{k-1}$ - {p, q | c = join(p, q)}
11. end
12. Answer1 = $U_k O_k$

Algorithm cc-gen(L_{k-1})
Two executions:
L_{k-1} p: p.item$_1$, p.item$_2$, …, p.item$_{k-1}$
L_{k-1} q: q.item$_1$, q.item$_2$, …, q.item$_{k-1}$
Join L_{k-1} p with L_{k-1} q to build L_k P: p • q.item$_{k-1}$
Insert into Ck
Select p.item$_1$, p.item$_2$, …., p.item$_{k-1,}$ q.item$_1$, q.item$_2$, …., q.item$_{k-1}$
From L_{k-1} p, L_{k-1} q
Where p.item$_2$ = q.item$_1$,….p.item$_{k-1}$ = q.item$_{k-2}$

3. BENCHMARK PROGRAMS AND ASSESSMENT

In order to see if any particular behaviors exit in typical Java programs, we collected a suite of Java programs to analyze. These programs were first executed on the modified JVM to get the run-time method invocation sequence and then analyzed using the analyzer to obtain various statistics. In this section, the benchmark programs are described and the results are discussed.

3.1 Benchmark Programs

We collected 18 Java programs for our analyzer to analyze. Most of these programs came from two sources. One was sample programs included in the JDK, and the other was winning programs from the JavaCup program contest, which was held by Sun Microsystems in 1996. The Javac program is included in the JDK API. LinpackJava was downloaded from [16]. It was hoped that these programs could represent the application domains of Java programs and exhibit typical java program behaviors.

In the following, we describe our benchmark programs. In the **# of Classes** field, the number in parentheses is the number of classes that exist in the program, while the number outside the parentheses is the number of classes that are actually used during program execution. Most of these programs are user-intervention programs. In other words, users need to terminate execution of these programs. We always terminated their execution after the execution behaviors had reached a steady state. For example, in the case of the

Animation program, we kept the program running so that the animation repeated two or three times before terminating it. In the case of the WebDraw program, we drew a Mickey Mouse face and saved it before exiting the program.

These benchmark programs can be classified into the following eight categories:

1. **Text Processing:** Javac
2. **Image Processing:** Animation, MoleculeViewer, ScrollText, Blink, Fractal, DitherT-est
3. **Game:** TicTacTo, Tubes.
4. **Multi-Thread Program:** BackgroundThread, ThreadX.
5. **Interactive Program:** CardTes, MapInfo
6. **Simulation:** TrafficSim, TuringMachine
7. **System:** WebDraw, DigSim
8. **CPU Intensive program:** LinpackJava

Table 1. An overview of the benchmark programs used in this study.

Name	# of Lines	# of Classes	# of Events
Javac	2,570	156(8)	272,193
Animation	361	139(1)	70,942
MoleculeViewer	705	132(4)	558,202
ScrollText	307	121(1)	32,907
Blink	94	111(1)	59,977
Fractal	385	115(4)	134,158
DitherTest	332	141(3)	303,727
TicTacToe	306	146(1)	40,391
Tubes	617	149(8)	585,210
Background Thread	367	135(5)	159,120
ThreadX	278	118(3)	74,449
CardTest	113	118(2)	31,547
MapInfo	4,277	192(26)	306,904
TrafficSim	669	125(6)	563,661
TuringMachine	991	167(1)	156,045
WebDraw	5,170	156(23)	248,353
DigSim	10,293	225(64)	993,350
LinpackJava	629	39(1)	11,180

3.2 Runtime Statistics

Statistics obtained from several benchmark programs showed that control patterns did exist. The following sections will describe several particular situations.

3.2.1 Statistics - control patterns (Animation)

Table 2 reports the results of the Animation benchmark. It shows that control patterns did exist. Simple patterns, compound patterns, and complex patterns had different ratios. In this table, the CP and LP2[CP, S1] percentages are relatively high.

Table 2. Statistics of animation.

Animation		
Types	**The number of event**	**Percentage**
Simple pattern	**29547**	**42%**
Sequence(S)	6613	9%
CP	17574	25%
LP2	5360	8%
Compound pattern	**34750**	**49%**
S[S1,LP2]	989	1%
S[CP,LP2]	2867	4%
S[CP,CP,S1]	607	1%
S[CP,LP2,CP]	2685	4%
S[S2,CP,S1]	566	1%
S[S1,CP,S2]	490	1%
S[CP,CP,S2]	698	1%
LP3[CP,S3,CP]	2185	3%
LP2[CP,CP]	5036	7%
LP2[CP,S1]	13240	19%
LP2[S1,CP]	453	1%
LP3[CP,CP,LP2]	4934	7%
Complex pattern	**6645**	**9%**
Total number of event	**70942**	**100%**

3.2.2 Statistics - control patterns (LinpackJava)

Table 3 lists the statistics for LinpackJava. It is interesting that the CP patterns represented 96 percent of the patterns. LinpackJava is a kind of kernel benchmark. It contains a big LOOP for a specific pattern repeatedly.

3.2.3 Statistics - control patterns (BackgroundThread & DitherTest)

The object-oriented program proceeded with object invocations conducted one at a time. The object invocation sequence can be thought of as a large sequence pattern, which can also be regarded as a trivial pattern. In Table 4, BackgroundTread has a particular LP3 patterns that is 65 percent. The S(CP, CP) of DitherTest is 76 percent as shown in Table 5.

Table 3. Statistics of LinapckJava.

LinpackJava		
Types	**The number of event**	**Percentage**
Simple pattern	**11098**	**99%**
Sequence(S)	101	1%
CP	10787	96%
LP2	0	0%
LP3	210	2%
Compound pattern	**82**	**1%**
LP2[CP,CP]	72	1%
LP2[S1,CP]	10	0%
Complex pattern	**0**	**0%**
Total number of event	**11180**	**100%**

Table 4. Statistics of BackgroundThread.

BackgroundThread		
Types	**The number of event**	**percentage**
Simple pattern	**106663**	**67%**
Sequence(S)	554	0%
CP	2861	2%
LP2	408	0%
LP3	102840	65%
Compound pattern	**52457**	**33%**
S[S1,LP2]	3512	2%
S[LP2,LP2]	5528	3%
S[CP,CP,CP]	1632	1%
S[CP,S2,CP]	1176	1%
S[S2,CP,S2]	1332	1%
S[S2,LP2,CP]	21539	14%
LP2[CP,CP]	6283	4%
LP2[CP,S1]	4335	3%
LP3[CP,CP,LP2]	4874	3%
LP3[CP,CP,S1]	20	0%
LP3[CP,LP2,S1]	2226	1%
Complex pattern	**0**	**0%**
Total number of event	**159120**	**100%**

3.2.4 Statistics - control patterns (TuringMachine)

Table 6 shows the statistics for TuringMachine. TuringMachine has 32 kinds of control patterns, one with the highest numbers among our benchmark programs. This indicates that there are not only more kinds of control patterns but also more kind of

Table 5. Statistics of DitherTest.

DitherTest		
Types	**The number of event**	**percentage**
Simple pattern	**55059**	18%
Sequence(S)	0	0%
CP	54710	18%
LP2	235	0%
LP3	114	0%
Compound pattern	**248668**	**82%**
S[CP,CP]	231600	76%
LP2[CP,CP]	1510	0%
LP2[CP,S1]	10590	3%
LP2[S1,CP]	94	0%
LP3[CP,CP,LP2]	4874	2%
Complex pattern	**0**	**0%**
Total number of event	**303727**	**100%**

complex behaviors. Different programs have different combinations of behaviors. It is impossible to show all of them here in the statistics. In addition, they all have their own behaviors. Tables A-J in Appendix A provide detailed information regarding the percentages of different patterns in the rest of the benchmark programs.

3.2.5 Other statistics

The statistics of other benchmark program corpora are listed in Appendix A.

3.3 Comparison

From the experimental results of the benchmark programs, we found that not all the programs have all three kinds of patterns. In Fig. 11, three values are shown for each program.

The first value is the percentage of complex patterns, which is listed here for comparison with the other two values. The second value is the percentage of compound patterns, and the third is the percentage of Simple pattern.

Fig. 12 shows the numbers of different types of patterns, including simple, compound and complex patterns. The higher the number, the more complex the behavior is expressed. For example, LinpackJava and DitherTest are simpler. TuringManchine and Javac are more complex.

Most of the programs have nontrivial behavior. In Fig. 13, only TuringMachine is higher than thirty percent. The values of Mapinfo and WebDraw are more than ten percent. The rest of the programs are below ten percent.

Fig. 14 shows that there are more programs with respect to particular patterns.

Table 6. Statistics of TuringMachine.

TuringMachine		
Type	**The number of events**	**Percentage**
Simple patterns	**49652**	**32%**
Sequence(S)	6643	4%
CP	23756	15%
LP2	19217	12%
LP3	36	0%
Compound patterns	**106393**	**68%**
S[S1,CP]	1132	1%
S[S1,CP,CP]	1050	1%
S[CP,S1,CP]	1379	1%
S[CP,S2]	568	0%
S[LP2,CP,S1]	3031	2%
S[LP2,LP2,CP]	5434	3%
S[S1,CP,LP2,CP]	4655	3%
S[S1,LP2,CP,CP]	3479	2%
S[LP2,CP,CP,S1]	3822	2%
S[S1,CP,S3]	2744	2%
S[CP,LP2,CP,CP,CP]	4425	3%
S[CP,S3,LP2]	4347	3%
S[CP,S3.CP]	1404	1%
S[CP,S5]	1192	1%
S[CP,CP,S4]	1640	1%
S[CP,CP,CP,S1,LP2,CP]	4605	3%
S[LP2,CP,S4]	3699	2%
S[LP2,CP,S3,CP]	4107	3%
S[S3,CP,S3,LP2]	3975	3%
S[CP,S3,CP,S4,CP]	4016	3%
S[S1,CP,CP,S1,CP,CP,CP,S1]	5076	3%
S[CP,CP,S1,CP,CP,CP,S2,LP2]	5005	3%
LP3[S2,LP2]	1496	1%
LP2[CP,CP]	7407	5%
LP2[CP,S1]	14933	10%
LP2[S1,CP]	620	0%
LP3[CP,CP,LP2]	4894	3%
LP3[CP,LP2,S1]	6258	4%
Complex patterns	**0**	**0%**
Total number of events	**156045**	**100%**

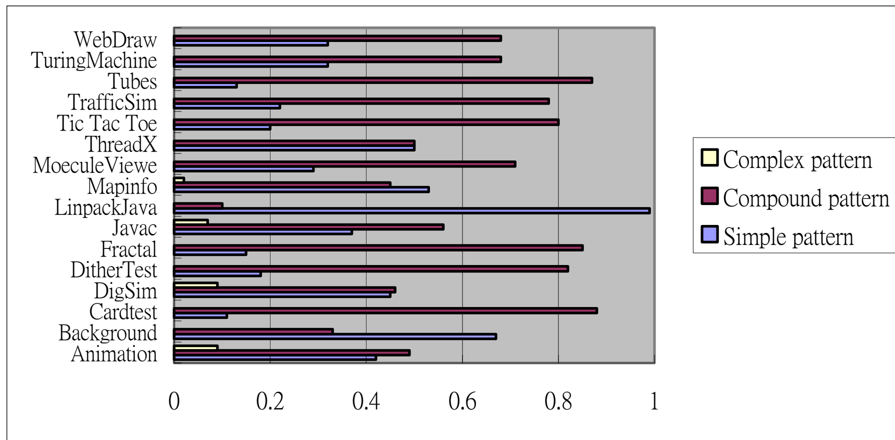

Fig. 11. Percentages of complex patterns, compound patterns, and simple patterns.

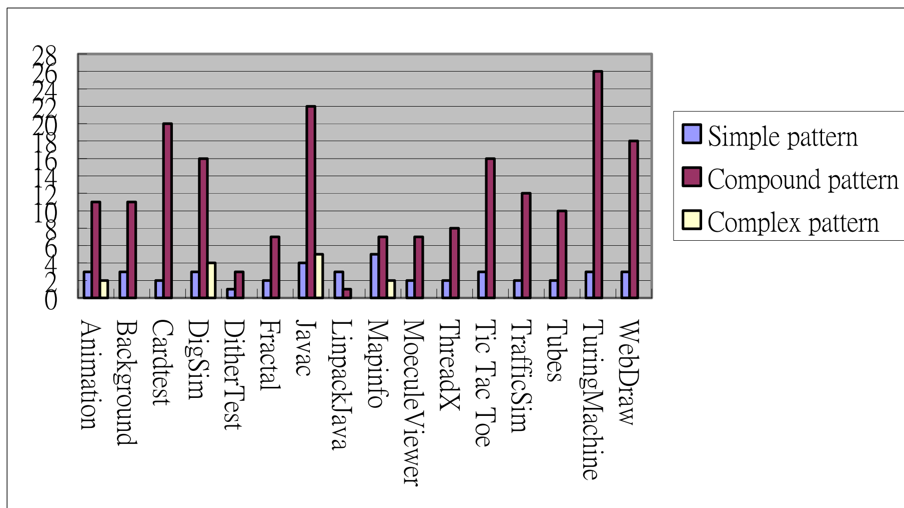

Fig. 12. Numbers of complex patterns, compound patterns, and simple patterns.

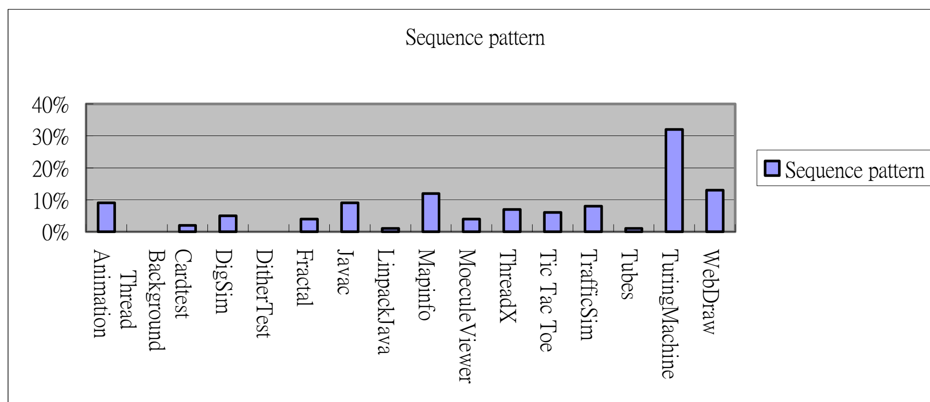

Fig. 13. Percentages of sequence patterns.

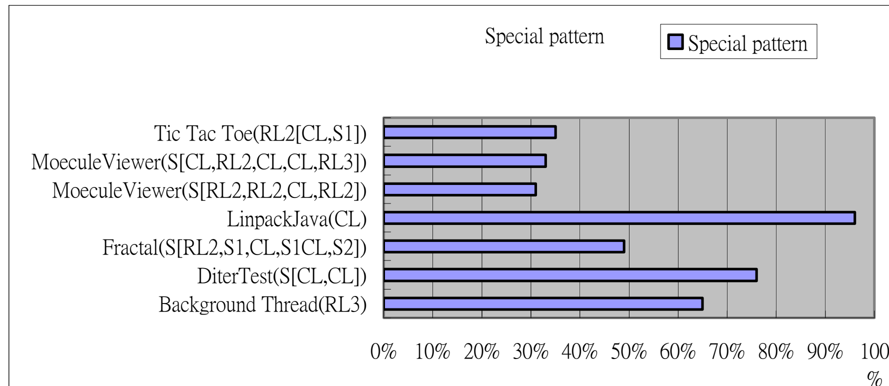

Fig. 14. Percentages of specific patterns.

4. CONCLUSIONS

We collected 18 Java programs and used them as benchmark programs in our study on control patterns. These 18 Java programs can be grouped into eight different application categories. After obtaining the run-time information of these benchmark programs, we used our analyzer to analyze the method size, native method percentages, method invocation localities, and control patterns in these program corpora.

We modified the JVM implemented by Sun Microsystems to collect method invocation sequences and run time log file information for the purpose of control pattern analysis. In this paper, several control patterns have been proposed and discussed. In particular, we have analyzed and collected several control patterns from several Java program corpora. The experimental results show that control patterns do exist. Simple pattern, compound patterns and complex patterns have different ratios depending on their source programs. Not all the benchmark programs contain all three kinds of patterns.

These control patterns explains which control jump occurs more frequently in the static class hierarchy. The call graph and static class hierarchy tells the associations among classes in the original programs. But it cannot tell the information as to which part is the bottleneck. The bottleneck of the object-oriented programs can be found from the runtime behaviors, which can be shown indirectly from these control patterns statistics. Furthermore, these control patterns can be used as guidelines for programmers to explain the run time behaviors and thus, to avoid possible run time overheads by redesigning their programs.

REFERENCES

1. S. K. Huang, “Optimizing run-time behaviors in object-oriented programming systems,” Ph.D. Dissertation, Institute of Computer Science and Information Engineering, National Chiao Tung University, 1996.
2. C. C. Hwang, “Object-oriented program behavior analysis based on control patterns,” Ph.D. Dissertation, Institute of Computer Science and Information Engineering, Na-

tional Chiao Tung University, 2002.
3. I. Jacobson, *Object-oriented Software Engineering*, Addison-Wesley, 1992..
4. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall Inc., 1991.
5. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Pub. Co., 1997.
6. C. C. Hwang, S. K. Huang, D. J. Chen, and M. S. Lin, "Dynamic java program corpus analysis part1: the analyzer," *Journal of Object-Oriented Programming*, Vol. 14, 2001, pp. 26-29.
7. J. Vitek, R. N. Horspool, and A. Krall, "Efficient type inclusion tests," in *Proceedings of the Conference on Object Oriented Programming Systems*, *Languages & Applications* (*OOPSLA '97*), 1997, pp. 142-157.
8. A. Krall, J. Vitek, and N. Horspool, "Near optimal hierarchical encoding of types," in *Proceedings of 11th European Conference on Object-Oriented Programming* (*ECOOP '97*), 1997, pp. 128-145.
9. J. Dean, D. Grove, and C. Cambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of 9th European Conference on Object-Oriented Programming* (*ECOOP '95*), 1995, pp. 77-101.
10. P. Michaud, "Clustering techniques," *Future Generation Computer Systems*, 1997, pp. 135-147.
11. M. R. Anderberg, *Cluster Analysis for Applications*, Academic Press, 1973.
12. C. C. Hwang, S. K. Huang, M. S. Lin, and D. J. Chen, "Dynamic java programming corpus analysis part2: the control pattern analysis," *Journal of Object-Oriented Programming*, Vol. 14, 2001, pp. 17-23.
13. C. C. Hwang, S. K. Huang, D. J. Chen, and D. T. K. Chen, "Object-oriented program behavior analysis based on control patterns," in *Proceedings of the 2nd Asia-Pacific Conference on Quality Software*, 2001, pp. 81-87.
14. G. Booch, *Object-oriented Analysis and Design with Applications*, Benjamin Cummings, 1994.
15. G. Booch, J. Rumbaugh, et. al., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, 1999.
16. J. Dongarra, R. Wade, and P. McMahan, *LinpackJava*, http://www.netlib.org/benchmark/linpackjava/, 2002.

APPENDIX A

Table A. Control pattern distribution in % (DigSim).

DigSim		
Type	**The number of events**	**Percentage**
Simple patterns	**70129**	**45%**
Sequence(S)	7826	5%
CL	34250	22%
RL2	28017	18%
RL3	36	0%

Compound patterns	**72468**	**46%**
JP[(A,B),CL]	1201	1%
S[RL2,RL2]	4935	3%
S[S1,CL,CL]	1080	1%
S[CL,S1,CL]	1418	1%
S[CL,S2]	724	0%
S[RL2,CL,S1]	3045	2%
S[CL,S3,CL,S1]	1591	1%
S[CL,CL,S1,CL,S2]	1980	1%
S[CL,CL,CL,S1.RL2.CL]	4666	3%
S[S1,CL,RL2,CL]	4693	3%
S[CL,S3]	1047	1%
RL2[S2,RL2]	1496	1%
RL2[CL,CL]	10788	7%
RL2[CL,S1]	21749	14%
RL2[S1,CL]	903	1%
RL3[CL,CL,RL2]	4894	3%
RL3[CL,RL2,S1]	6258	4%
Complex patterns	**13448**	**9%**
Total number of events	**156045**	**100%**

Table B. Control pattern distribution in % (Fractal).

Fractal		
Type	**The number of events**	**Percentage**
Simple patterns	**19618**	**15%**
Sequence(S)	4700	4%
CL	14839	11%
RL2	61	0%
RL3	18	0%
Compound patterns	**114540**	**85%**
S[RL2,CL,S3,CL,S1,CL,S3]	30240	23%
S[RL2,S1,CL,S1,CL,S2]	65208	49%
RL3[CL,S3,CL]	2185	2%
RL2[CL,S2]	48	0%
RL2[CL,CL]	304	0%
RL2[CL,S1]	8428	6%
RL2[S1,CL]	2805	2%
RL3[CL,CL,RL2]	4874	4%
RL3[CL,RL2,S1]	448	0%
Complex patterns	**0**	**0%**
Total number of events	**134158**	**100%**

Table C. Control pattern distribution in % (Javac).

Javac		
Type	**The number of events**	**Percentage**
Simple patterns	**101698**	**37%**
Sequence(S)	24181	9%
CL	44331	16%
RL2	20331	7%
RL3	12855	5%
Compound patterns	**151433**	**56%**
JP[(A,B,C,D,E),CL]	4617	2%
DP[CL,(A,B,C)]	2586	1%
S[S1,CL,CL]	1452	1%
S[CL,S2]	1283	0%
S[CL,S1,CL]	1696	1%
S[CL,S2]	1226	0%
S[CL,CL,S1]	1561	1%
S[CL,CL,RL2]	7464	3%
S[CL,RL2,S1]	7300	3%
S[RL2,RL2,S1]	6425	2%
S[S1,CL,CL,S1]	1790	1%
S[S1,CL,S2]	3260	1%
S[S2,CL,S1]	1550	1%
S[S2,CL,CL]	2279	1%
S[CL,S2,CL]	1716	1%
S[RL2,CL,S2]	36372	13%
S[RL2,CL,S5,CL]	29926	11%
RL2[CL,S2]	8160	3%
RL2[S2,CL]	6264	2%
RL2[S2,RL2]	1709	1%
RL2[CL,CL]	991	0%
RL2[S1,CL]	10182	4%
RL2[CL,S1]	11624	4%
Complex patterns	**19062**	**7%**
Total number of events	**272193**	**100%**

Table D. Control pattern distribution in % (Mapinfo).

Mapinfo		
Type	**The number of events**	**Percentage**
Simple patterns	**163696**	**53%**
Sequence(S)	36027	12%
CL	78906	26%
RL2	2736	1%

RL3	45027	15%
JP	1000	0%
Compound patterns	**137468**	**45%**
S[RL2,RL2,S2,CL]	5208	2%
RL3[S2,RL2]	9900	3%
RL2[S2,CL]	2152	1%
RL2[CL,CL]	18114	6%
RL2[CL,S1]	92754	30%
RL2[S1,CL]	4286	1%
RL3[CL,CL,RL2]	5054	2%
Complex patterns	**5740**	**2%**
Total number of events	**306904**	**100%**

Table E. Control patterns distribution in % (MoleculeViewer).

MoleculeViewer		
Type	**The number of events**	**Percentage**
Simple patterns	**159126**	**29%**
Sequence(S)	23704	4%
CL	135104	24%
RL2	300	0%
RL3	18	0%
Compound patterns	**399076**	**71%**
S[CL,RL2,CL,CL,RL2]	175516	31%
S[RL2,RL2,CL,RL2]	184852	33%
RL3[S2,RL2]	9900	2%
RL2[S2,CL]	2152	0%
RL2[CL,CL]	3900	1%
RL2[CL,S1]	11635	2%
RL2[S1,CL]	4993	1%
RL3[CL,CL,RL2]	4874	1%
RL3[S2,RL2]	1254	0%
Complex patterns	**0**	**0%**
Total number of events	**558202**	**100%**

Table F. Control patterns distribution in % (ThreadX).

ThreadX		
Type	**The number of events**	**Percentage**
Simple patterns	**36962**	50%
Sequence(S)	27621	37%
CL	9013	12%
RL2	220	0%
RL3	108	0%

Compound patterns	**37487**	**50%**
S[RL2,RL2]	3414	5%
S[CL,RL2,CL,S1]	4592	6%
S[S3,CL,CL]	1444	2%
S[CL,CL,CL]	3329	4%
S[RL2,CL]	8066	11%
RL2[CL,CL]	2999	4%
RL2[CL,S1]	8720	12%
RL2[S1,CL]	51	0%
RL3[CL,CL,RL2]	4872	7%
Complex patterns	**0**	**0%**
Total number of events	**74449**	**100%**

Table G. Control patterns distribution in % (Tic Tac Toe).

TicTacToe		
Type	**The number of events**	**Percentage**
Simple patterns	**8200**	**20%**
Sequence(S)	2540	6%
CL	5263	13%
RL2	346	1%
RL3	51	0%
Compound patterns	**32191**	**80%**
S[S1,RL2]	1225	3%
S[CL,S1]	184	0%
S[S1,CL,CL]	385	1%
S[S1,RL2,S1]	1054	3%
S[CL,S2]	255	1%
S[CL,CL,S1]	343	1%
S[CL,CL,CL]	1305	3%
S[CL,CL,RL2]	1102	3%
S[RL2,CL,CL]	1642	4%
S[CL,S2,CL]	408	1%
S[S2,CL,S2]	329	1%
S[S3,CL,S1]	1029	3%
S[S1,CL,CL,S2]	4212	10%
RL3[S2,RL2]	1650	4%
RL2[CL,CL]	2305	6%
RL2[CL,S1]	14258	35%
RL2[S1,CL]	373	1%
RL3[RL2,CL,S1]	132	0%
Complex patterns	**0**	**0%**
Total number of events	**40391**	**100%**

Table H. Control patterns distribution in % (TrafficSim).

TrafficSim		
Type	**The number of events**	**Percentage**
Simple patterns	**125295**	**22%**
Sequence(S)	46100	8%
CL	78953	14%
RL2	188	0%
RL3	54	0%
Compound patterns	**438366**	**78%**
S[S1,RL2]	26858	5%
S[RL2,S1]	26728	5%
S[CL,CL,CL]	9180	2%
S[CL,CL,RL2,CL,CL]	33300	6%
S[CL,CL,S1,CL,CL]	13403	2%
S[RL2,S1,RL2]	52581	9%
S[CL,CL,S2,RL2]	33957	6%
S[CL,CL,S1,RL2]	32928	6%
RL2[CL,CL]	53121	9%
RL2[CL,S1]	83023	15%
RL2[S1,CL]	68413	12%
RL3[CL,CL,RL2]	4874	1%
Complex patterns	**0**	**0%**
Total number of events	**563661**	**100%**

Table I. Control patterns distribution in % (Tubes).

Tubes		
Type	**The number of events**	**Percentage**
Simple patterns	**76530**	**13%**
Sequence(S)	8343	1%
CL	65350	11%
RL2	2741	0%
RL3	96	0%
Compound patterns	**508680**	**87%**
S[CL,RL2]	86496	15%
S[RL2,CL]	79866	14%
S[CL,S1,CL]	24948	4%
S[CL,CL,S1,CL,S1]	43008	7%
S[RL2,RL2,S1,CL,S1,CL,S1,CL]	139378	24%
S[CL,CL,S1,S1,CL,S1]	97240	17%
RL2[S2,CL]	2152	0%
RL2[CL,CL]	1278	0%
RL2[CL,S1]	8033	1%

RL2[S1,CL]	19656	3%
RL3[CL,CL,RL2]	5104	1%
RL3[RL2,RL2,S1]	999	0%
RL3[RL2,RL2,RL2]	522	0%
Complex patterns	**0**	**0%**
Total number of events	**585210**	**100%**

Table J. Control patterns distribution in % (WebDraw).

WebDraw		
Type	**The number of events**	**Percentage**
Simple patterns	**79925**	**32%**
Sequence(S)	31321	13%
CL	43329	17%
RL2	850	0%
RL3	4425	2%
Compound patterns	**168428**	**68%**
S[S1,RL2]	3288	1%
S[S1,CL]	716	0%
S[S1,RL2,S1]	3836	2%
S[CL,S1,CL]	1743	1%
S[CL,CL,S2]	1552	1%
S[CL,S4]	1554	1%
S[S3,CL,S1,CL,CL,CL]	3136	1%
S[S2,CL,S1,CL,S2,CL]	2744	1%
S[S4,CL,S2,CL,CL,CL]	6372	3%
S[CL,RL2,CL,CL,RL2]	31320	13%
S[CL,CL,S1,CL,RL2,S1]	12980	5%
S[S2,CL,S2]	4130	2%
S[CL,CL,CL]	3186	1%
S[S1,CL,S1]	1770	1%
RL2[CL,S2]	1584	1%
RL2[CL,CL]	32238	13%
RL2[CL,S1]	46804	19%
RL2[S1,CL]	2411	1%
RL3[CL,CL,RL2]	4874	2%
RL3[CL,RL2,S1]	308	0%
RL3[RL2,S1,CL]	630	0%
RL3[S1,CL,CL]	340	0%
RL3[S1,RL2,CL]	912	0%
Complex patterns	**0**	**0%**
Total number of events	**248353**	**100%**

Deng-Jyi Chen (陳登吉) received the B.S. degree in Computer Science from Missouri State University (cape Girardeau), U.S.A., and M.S. and Ph.D. degree in Computer Science from the University of Texas (Arlington), U.S.A. in 1983, 1985,1988, respectively. He is now a professor at Computer Science and Information Engineering Department of National Chiao Tung University (Hsinchu, Taiwan). Prior to joining the faculty of National Chiao Tung University, he was with National Cheng Kung University (Tainan, Taiwan). So far, he has been publishing more than 110 referred papers in the area of performance and reliability modeling and evaluation of distributed systems, computer networks, software reuse, object-oriented systems, and multimedia application systems. Some of his research results have been technology transferred to industrial sectors and used in product design and implementation. So far, he has been a chief project leader of several commercial products. Some of these products are widely used in primary schools for CAI educational tools in Taiwan. He has been received both research awards and teaching awards from various organizations in Taiwan and serves as a committee member in several academic and industrial organizations.

Chung-Chien Hwang (黃中見) was born on April 24, 1970 in Nan-Tou, Taiwan. He received the M.S. degree in 1995 and the Ph.D. degree in 2002, both from the Computer Science and Information Engineering at National Chiao-Tung University. He is presently a Chief Market Officer in Bestwise International Computing CO., LTD.. His research interests include object-oriented computing, multimedia authoring tools, video conference, and distance learning.

Shih-Kun Huang (黃世昆) received the B.S., M.S., and Ph.D. degrees in Computer Science and Information Engineering from National Chiao Tung University in 1989, 1991, and 1996, respectively. He joined the Institute of Information Science, Academia Sinica as an Assistant Research Fellow in 1997. His research interests include system security, rights management, and software engineering.

David T. K. Chen (陳廷楷) is a faculty member in the Department of Computer and Information Sciences at the Fordham University in New York. Prior to this position, he was with IBM Corporation working in the areas of systems analysis, marketing and research. His main research interests are in object-oriented database design, database modeling methodologies, software engineering, computer networking, and information security. David Chen received his Ph.D. in Information Science from the University of Erlangen-Nuremberg, Germany in 1973. He is a member of IEEE Computer Society, ACM, and The New York Academy of Sciences.