Note: This paper is submitted for the ICS 2002 at Hualien, Taiwan Dec. 11-14,2002. It contains so many tables and figures for reviewers' reference. We will condense the paper to meet program committee's requirement once the paper is accepted for publication. Thanks for your time and effort for processing our manuscript.

1) **Name of the workshop: Workshop on Databases and Software Engineering**
2) **Title of the paper: Control Patterns Analysis on Java Program Corpora**
3) **Authors:**

Chung-Chien Hwang, and Deng-Jyi Chen
Computer Science and Information Engineering Department,
National Chiao Tung University, Hsin-Chu, Taiwan
cchwang@csie.nctu.edu.tw, djchen@csie.nctu.edu.tw

Shih-Kun Huang
Institute of Information Science, Academia Sinica,
128 Academic Road, Section 2, Nankang 115, Taipei, Taiwan
skhuang@iis.sinica.edu.tw

David T. K. Chen
Computer and Information Science Department,
Fordham University, Bronx, N.Y. U.S.A.
dachen@fordham.edu

4) **Contact person:**

Professor Deng-Jyi Chen
Computer Science and Information Engineering Department,
National Chiao Tung University, Hsin-Chu, Taiwan

5) **Keywords: OOP, Control Patterns, Data Mining, Java VM, Code Patterns, Benchmark Design, Program Optimization, Static and Dynamic Analysis.**

6) Abstract:

Java programming, based on Object-Oriented (OO) paradigm, has played a major role in program design and implementation due to the fact that it is more extensible, maintainable, and reusable in the software system construction. Experiences of using Java programming have indicated that there exist disadvantages with respect to its execution inefficiency and complicated runtime behaviors. Program analysis is essential for performance measurement and improvement. Current static and dynamic analysis using OO programming cannot characterize runtime behavior well and are also hard to quantify the measured results. In this paper, research work was performed to analyze several Java program metrics and method invocation sequence. The results not only provide us a better understanding of the runtime behavior but also present more information for different application domains.

Code-patterns are statically recurring structure specifically related to a programming language. It can be used in parallel to help designing software systems for solving particular problems. In opposition to code-patterns' role in assisting compilation, control-patterns are dynamically recurring structures invoked during program execution time. It can be used to understand the run-time behaviors of OO-programs for the underlying architecture such as Java-VM. Control pattern describes the model of control transfer among objects in OO program execution. In this research, several control patterns are proposed and discussed. Particularly, we have analyzed and collected several control patterns over several Java program corpora. The experimental results show that control pattern does exist and provide quantitative analysis. Simple pattern, compound pattern and complex pattern have different ratio respectively, according to a variety of different source programs. Control patterns collected can be used to provide guidelines for Java programmers to write more effective Java program.

1. Introduction

Over the years, issues relating to performance improvement in the run-time environment of OO systems have been studied and researched in literatures [1,2,6,9,12]. Since the run time behaviors with respect to various application domains and code patterns are different substantially, the optimization should be domain or pattern specific [1]. Programs designed and written by object-oriented approach are more extensible, maintainable and reusable than those produced by traditional procedural-oriented approach [3,4,14,15]. But, a significant increment of complex runtime behaviors has become the disadvantage associated with object-oriented design. The traditional concept of program analysis is not broad enough to represent the real runtime behavior of the OO program. 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 Java, there are only two types of control flows: sequential execution and message sending. Sequential execution is not so different from procedure-oriented program. The instructions are executed one by one without jumping to other places. Message sending means that the execution is switched to other groups of instructions. During the object-oriented program execution, the action of invoking a method to execute is known as a control transfer. A method invocation sequence records all of the control transfer, which is taking place during the execution of program. There might exhibit 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 runtime real world. With this in mind, the goal is to get the runtime method invocation sequence of object-oriented program execution first and build tools to analyze the invocation sequence to better understand the property of the runtime behavior. Then, the control pattern will be found based on the analyzed patterns. Thus, the performance measurement and performance improvement based on the control patterns can be measured hereafter.

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

2. Control Patterns

The progress of object-oriented program is regarded as the lifetime of objects in program execution. During the execution of object-oriented program, the action of invoking a method to execute is known as a control transfer. A method invocation sequence keeps track of all the control transfers occurred during program execution. Although the real action of method invocation is the control transfer among objects, in terms of the behavior of program execution, we can transform the control transfer into different kind of transfer between receiver classes. A program of call graph represents the possible callees at each call site in each procedure. Interprocedural analyses typically produce summary results of 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 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 from the method invocation sequence is more concrete than call graph.

Control pattern includes a directed graph that is a small subgraph of call graph plus two functions: constraint output function and constraint Boolean function. It can explain which subgraph is really executed in call graph as well as the quantitative results of this subgraph. Moreover, constraint output function and constraint Boolean function describe the real execution path in subgraph. Besides, class hierarchy analysis [7-9] exploits information about the structure of the class inheritance graph. Execution log pattern identifies the movement of control in class inheritance graph. It is possible that the frequency of dynamically message binding is reduced by split and combination of class. A method invocation sequence records all the control transfers occurred during program execution. It is the same meaning that a program segment is run with specific input data. A control pattern is a repeating pattern extracted from method invocation sequence.

2.1 Semantic meanings of control patterns

We have evaluated several kinds of simple control pattern (consecutive patterns, loop-N patterns [12], sequence pattern, dispatch pattern and join pattern [13]) according to language feature. A formal definition of the 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 pattern. Compound control pattern is a combination of simple control pattern. Control patterns of other specific appearances are defined as complex control pattern. Like a sequential statement will be executed sequentially, there are some program segment in OO program may be executed in the same manner. We call them sequence pattern. The semantic meanings of these three groups of control pattern are introduced in the following.

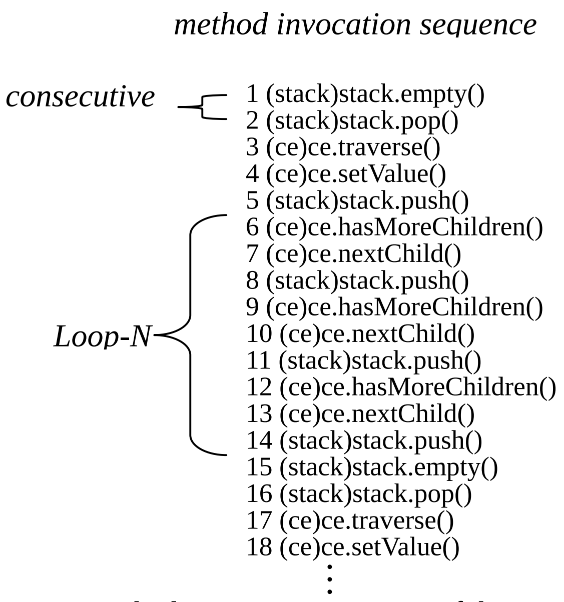
2.1.1 The semantic meanings of simple control pattern

Take the segment of program in Figure 2-1 as an example. It traverses a tree and gives each node a number to represent its traversal order. In spite of how many classes and methods are shown in the whole programs, 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.

A method invocation is consisted of three parts: receiver class, method class, and method. Figure 2-2 shows the method invocation sequence produced by running the program segment of Figure 2-1. Consider the method invocation sequence in Figure 2-2. The 1st and the 2nd method invocation are instances of consecutive pattern because they are consecutive and their receiver classes are the same (stack). “CP” is the abbreviation of “Consecutive Pattern”. Consider the method invocation from the 6th to the 14th, the same method invocation is repeated for every three-method invocations. As a result, they are treated as an instance of Loop-3 pattern, abbreviated as “LP3.” Directed graphs show the concept of LP3 in Figure4-3.

```
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() );
   }
}
```

Figure2-1 Segment of a Tree Traversal Program



```
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()
        ⋮
```

Figure 2-2 Method Invocation Sequence of the Program Segment in Figure 2-1

Let’s look at another example shown in Figure2-4. On the left part of this figure is the definition of classes, whereas in the middle part it is a program segment. Also, on the right part is the corresponding method invocation sequence. Considering the first to the 4th method invocations, they produce two execution log patterns (AB, AC). As a result, they form an instance of dispatch pattern (DP) as shown in Figure 2-5. The 5th to the 8th method invocations (BD, CD) form an instance of join pattern (JP) as shown in Figure 2-6.

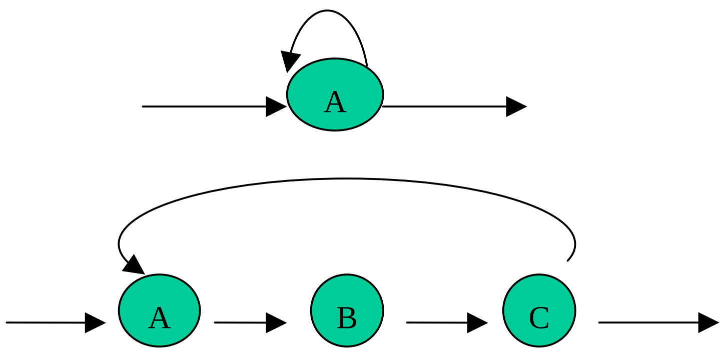


Figure 2-3 Consecutive Pattern and Loop-3 Pattern

```
CPass A {
        p1( ) {
        }
}

CPass B {
        m1( ) {
        }
}

CPass C extends B {
        m1( ) {
        }
}



CPass D {
```

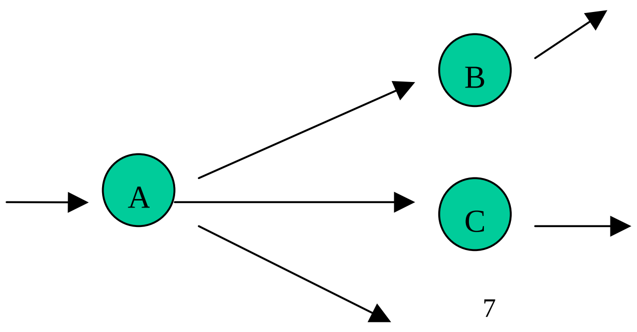
```
main( ) {
    CPass B v;
    CPass B x = new B( );
    CPass C y = new C( );
    CPass A u = new A( );
    CPass D z = new D( );

for( i=0;i<2;i++) {
            u.p( );
        if( i%2 == 0 )
            v = x;
        else
            v = y;
        v.m1( );       }
    for( i=0;i<2;i++) {
        if( i%2 == 0 )
            v = x;
        else
            v = y;
        v.m1( );
        z.q1;    }
}
```

```
(A)A.p1
(B)B.m1
(A)A.p1
(C)C.m1


(B)B.m1
(D)D.q1
(C)C.m1
(D)D.q1
```

Figure 2-4 Dynamic Message Sending Examples



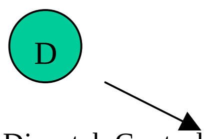


Figure 2-5 Dispatch Control Pattern

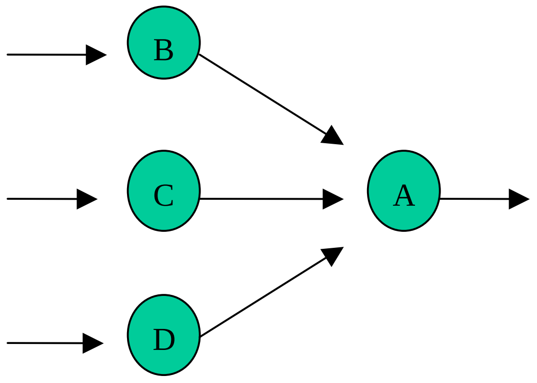


Figure 2-6 Join Control Pattern

2.1.2 Compound control pattern

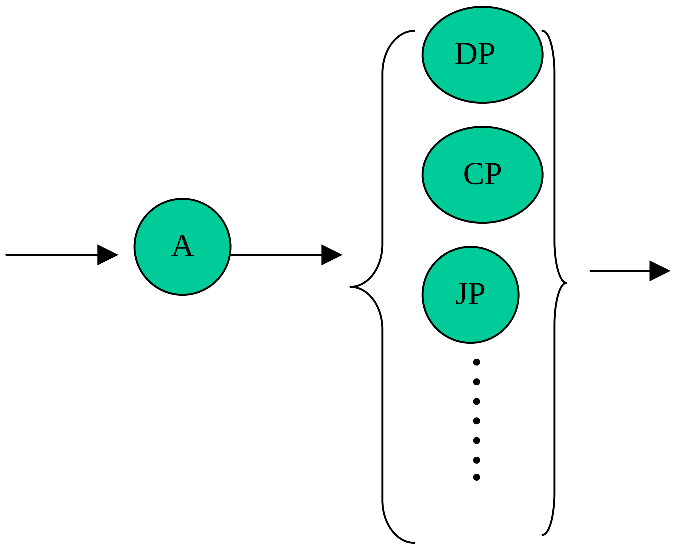


Figure 2-7 Example of Compound Control Pattern

A compound control pattern with G (E,V) is an isomorphs of simple control pattern with G'(E',V'). For each vertex of V', there can be either a compound control pattern or simple control pattern. Compound control pattern has architecture similar to a simple control pattern. The vertex of simple control patterns can be either a simple control pattern or compound control pattern. Figure 2-7 explains that the main skeleton of control pattern belongs to sequence pattern with 2 elements, and the second element can be either dispatch pattern, consecutive pattern, join pattern, and/or other compound control pattern.

2.1.3 Complex control pattern

In general, the directed graph is constructed for a set of execution log pattern. Every log pattern exists in the graph. Nevertheless, the directed graph doesn't belong to either a simple control pattern or a compound control pattern. A set of execution log pattern {ACDE, ABE, ACE} in Figure 2-8 can be used as an example.

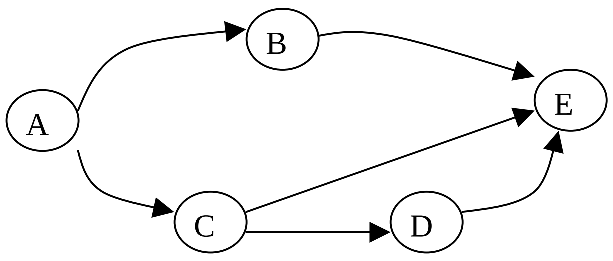


Figure 2-8 Example of Complex Control Pattern

In fact, a control pattern with G(E,V) is constructed based on a set S of execution log pattern. It is possible that a path from source to sink is in G but not in S. Considering a set of execution log pattern {AB_1CD_1E, AB_2CD_2E} in Figure 2-9, 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, there is a detailed definition of control pattern.

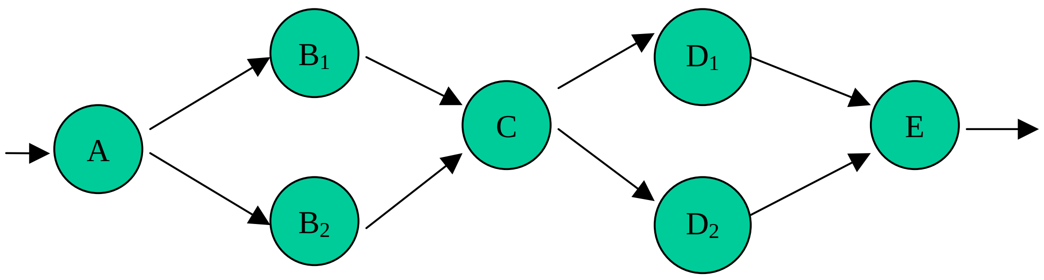


Figure 3-9 Example of Complex Control Pattern

2.2 Control patterns mining

When Java programs are executed on a modified JVM, a large database of events is given, where each event consists of receiver class, method class, method, event order, and the items bought into the event. All the events can together be viewed as a sequence, where each event corresponds to a set of items. And the list of event, labeled with increasing event order, corresponds to a sequence. In this section, an algorithm is described to solve the problem of mining sequential patterns over such 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, receiver class, method class, method, and event order. Each event order is a unique one. Mining patterns of execution log is the problem of finding a sequence of patterns concerned with an ordered list of method invocation. Mining sequence patterns can have many algorithms of implementation, 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 recurrence patterns in the method invocation sequence, and these recurrence patterns of control transfer are named as the execution log patterns.

Clustering analysis [10, 11] helps construct meaningful partitioning 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. It should possess small distances between elements of the same cluster and large distances between 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[xx] and is recalled here. One can divide the problem of control pattern mining into two parts. The first one is the graph construction and the second one 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 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 $E=\emptyset$.
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 the algorithm 1 is O(Σ 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)[15]:

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 $O=\{t|\ t|v|\alpha$ 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 $f_{(v,other\ vertices)}=\{0\}$
```

```
    else $f_{(v,v\text{-}next)}(o(v))=\{1\}$ and $f_{(v,other\ vertices)}= \{0\}$
8.  }
```

2.3 Mining execution log patterns

In this section, we describe how execution log patterns are looked up in a method invocation sequence. In fact, the concept of this problem is the same as mining sequence pattern. 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 control patterns identifier.

   ACD$\underline{\text{CCCCCCCC}}$EFHIAC$\underline{\text{ABCABCABCABCABC}}$KJEFDK

   $\Rightarrow$ ACD$\boxed{\text{CP}}$EFHIA$\boxed{\text{LP3}}$KJEFDK

The modified method invocation sequence is the input of sequence phase. Different simple control patterns are found separately. Only one control pattern is calculated for each pass of the method invocation sequence. Figure 3-10 [6] illustrates the method that our analyzer used to look up control patterns in a method invocation sequence. The method invocation sequence is fed into the predictor. The predictor keeps several internal states 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 is an instance of the evaluated control pattern. The input method invocation is then fed into the predictor to update the internal states of predictor.

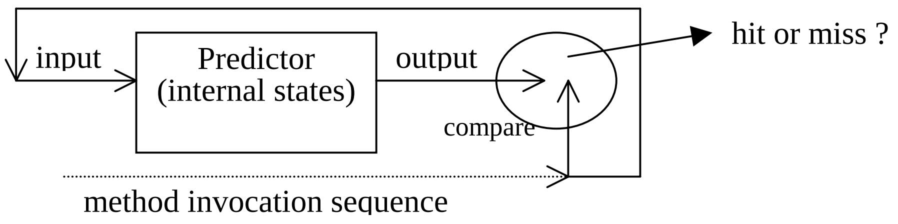


Figure 3-10 Predictor for Evaluating Control Patterns

The predictor is configurable. Setting up the predictor with different internal state configuration corresponds to different control pattern evaluation. Setting up the predictor with one internal state to record the previous method invocation is to evaluate the consecutive pattern. The internal state is used as an output to compare with next input method invocation. Setting up the predictor with three internal states to record the previous three method invocations is to evaluate the loop-3 pattern. The third internal state is used as an output to compare with the next input method invocation. By using these techniques, the consecutive pattern and loop-N pattern in a method invocation sequence can be easily found.

2. **Sequence Phase:** There are multiple passes over the modified method invocation sequence. In each pass, we start with a seed set of large sequences and call a sequence satisfying a minimal support constraint a large sequence. In the first pass, the seed set contains all 1-sequences with minimum support. The detail of the algorithms is described as follows.

```
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.      $L_k = \{c \in C_k | c.count \geq minsup\}$
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 $\bullet$ 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 there exist any particular behaviors in typical Java programs, we collected a suite of Java programs to analyze. These programs are first executed on the modified JVM to get the run-time method invocation sequence, and then analyzed by the analyzer to obtain various statistics. In this section, the benchmark programs are described and the results are discussed.

3.1 Benchmark Programs

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

Below are the overview and descriptions of our benchmark programs. In the **# of Classes** field, the number in the parentheses is the number of classes exist in the program, while the number outside the parentheses is the number of classes that are actually used in the run-time of the program execution. Most of these programs are user-intervention programs. In other words, it needs users to terminate the execution of these programs. We always terminate their execution after the execution behaviors have reached a steady state, or after proceeding a meaningful work. For example, in the Animation program, we kept the program running for the animation repeating two or three times before terminating it. In the WebDraw program, we drew a Mickey Mouse face and saved it before exiting the program.

Table 3-1 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

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

1. Text Processing: Javac

2. Image Processing: Animation,, MoleculeViewer, ScrollText, Blink, Fractal , DitherTest

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

3.2 Runtime statistics

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

3.2.1 Statistics - Control pattern (Animation)

Table 3-2 is a report of Animation benchmark. It shows that control pattern does exist. Simple pattern, Compound pattern and Complex pattern show different ratios. In this table percentages of CP and LP2[CP,S1] are relatively higher.

Table 3-2 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 pattern(LinpackJava)

Table 3-3 describes the statistic of LinpackJava. It has a small number of control patterns. It is interesting that CP pattern occupies 96 percent. LinpackJava is also greater than 90%. LinpackJava is a kind of kernel benchmark. It contains a big LOOP to run a specific pattern repeatedly. The percentage of CP pattern is thus very high.

Table 3-3 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%**

3.2.3 Statistics - Control pattern (BackgroundThread & DitherTest)

The progress of object-oriented program is proceeded with object invocations one at a time. The object invocation sequence can be thought of as a large sequence pattern, which is also regarded as a trivial pattern. The percentages of BackgroundThread and DitherTest are near 0%. It means that most of their control patterns are nontrivial patterns. Moreover, In Table 3-4, BackgroundTread has a particular pattern of LP3 which is 60 percent. The S(CP,CP) of DitherTest is 76 percent in Table 3-5.

.Table 3-4 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%**

Table 3-5 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%**

3.2.4 Statistics - Control pattern (TuringMachine)

Table 3-6 shows the statistics of TuringMachine. TuringMachine has 32 kinds of control patterns and it is the one with the highest number in our benchmark programs. It indicates not only more kinds of control pattern but also more complex behaviors. Different programs have various combination of behavior. It is impossible to show all of them in statistics. But they express their own behaviors. Tables A-J, in the appendix A, provide the detailed information regarding the percentage of different pattern for the rest of benchmark programs.

Table 3-6 Statistics of TuringMachine

TuringMachine		
Types	**The number of event**	**Percentage**
Simple pattern	**49652**	**32%**

Sequence(S)	6643	4%
CP	23756	15%
LP2	19217	12%
LP3	36	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**156045**	**100%**

3.2.5 Other Statistics

Appendix shows the statistics of other benchmark programs corpora.

3.4 Comparison

From the experimental results of benchmark programs, we found that not all programs have all three kinds of patterns. In Figure 3-1, three values are drawn for each program.

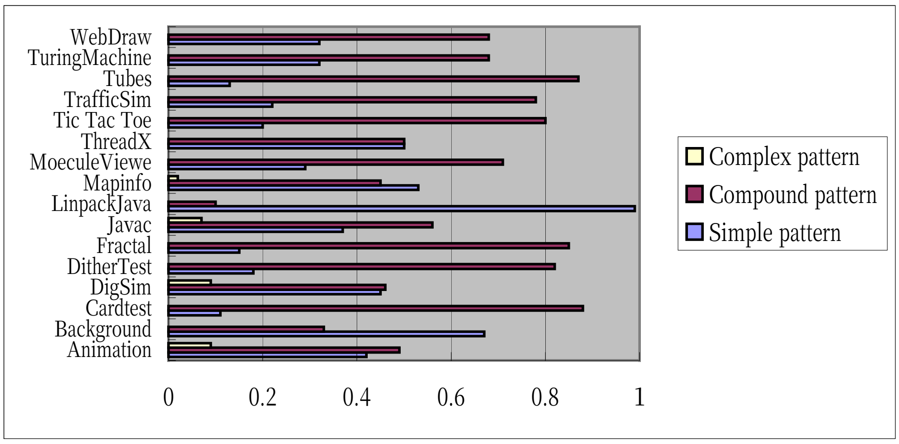


Figure 3-1 Percentages of Complex Pattern, Compound Pattern and Simple Pattern

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

Figure 3-2 shows the number of different types of patterns, including simple, compound and complex patterns. The greater the number, the more complex the behavior will be expressed. For example, LinpackJava and DitherTest are simpler. TuringManchine and Javac are more complex.

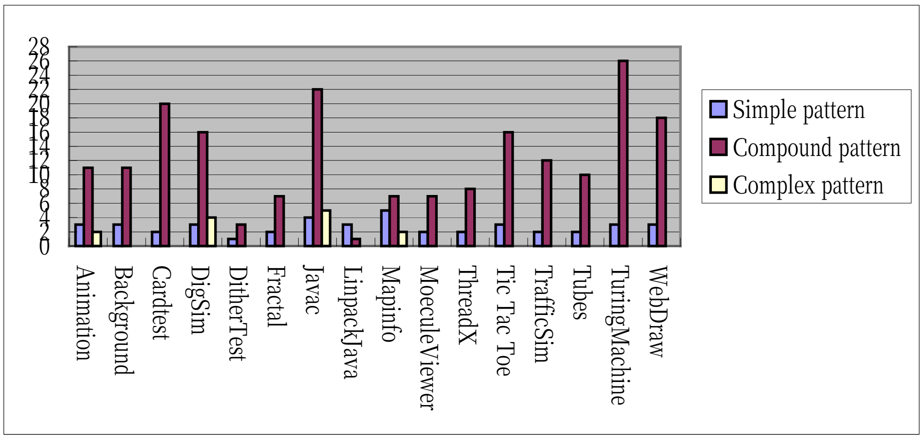


Figure 3-2 Numbers of Complex Pattern, Compound Pattern and Simple Pattern

Most programs have nontrivial behavior. In figure 3-3 only TuringMachine ’ s percentage is larger than thirty percent. The percentages of Mapinfo and WebDraw are larger than ten percent. The rest of the programs are below ten percent.

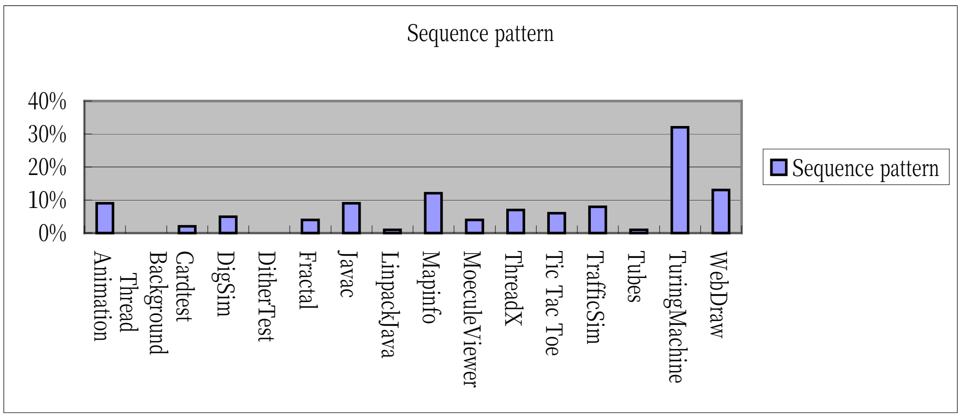


Figure 3-3 Percentages of Sequence Pattern

Figure 3-4 shows that there are more programs with respect to particular behaviors. The ratio is about 30 %. It can be seen that the percentage of BackgroundThread(LP3) is greater than 60%. DiterTest(S[CP.CP]) is near 80%. And the percentage of the remaining behaviors is also greater than 30 percent. These particular behaviors represent that the corresponding subgraphs appear more frequently on their call graph. At the same time, it also explains which control jump occurs more frequently on static class hierarch. Call graph and static class hierarch explain the relationships of class designed in the program. One is a calling relationship, and the other is an inherent relationship. They can ’ t provide the information as to which part is the bottleneck. The bottleneck of runtime behavior must be referred back to control pattern. Moreover, it plays an important role for programmers to redesign their program.

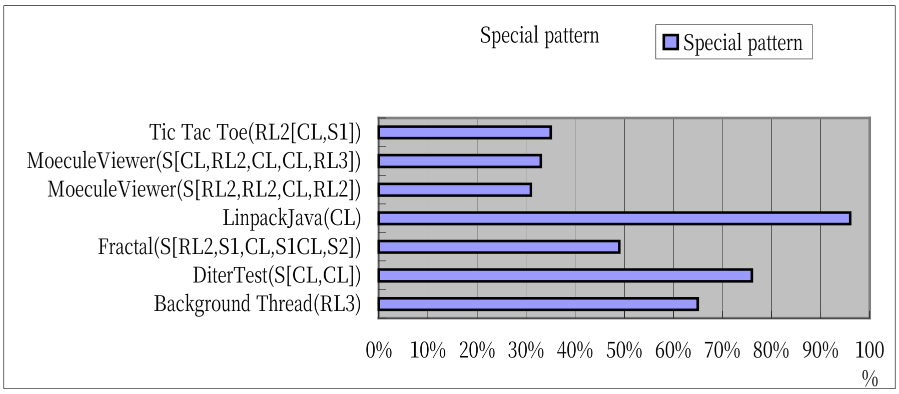


Figure 3-4 Percentages of Specific Behavior

4. Conclusion

We have collected 18 Java programs and used as our benchmark programs for control patterns study. These 18 Java programs can be grouped eight different application categories. After obtaining the run-time information of these benchmark programs, we use 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 sequence and the run time logfile information for control pattern analysis. In this research, several control patterns are proposed and discussed. Particularly, we have analyzed and collected several control patterns over several Java program corpora. The experimental results show that control pattern does exist and provide quantitative analysis. Simple pattern, compound pattern and complex pattern have different ratio respectively, according to a variety of different source programs. Not all benchmark programs contain all three kinds of patterns. There are high percentages of programs with nontrivial behaviors. The results not only provide us a better understanding of the runtime behavior but also present more information for different application domains.

5. References

[1] Shih-Kun Huang, *Optimizing Run-Time Behaviors in Object-Oriented Programming*

Systems, Ph.D. Dissertation of Institute of Computer Science and Information Engineering, National Chiao-Tung University, HsinChu, Taiwan, 1996.

[2] Chung-Chien Hwang, *Object-oriented Program Behavior Analysis Based on Control Patterns*, Ph.D. Dissertation of Institute of Computer Science and Information Engineering, National Chiao-Tung University, HsinChu, Taiwan, 2002.

[3] Ivan Jacobson, *Object-oriented Software Engineering*, Addison-Wesley, 1992..

[4] James Rumbaugh, Micheal Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall Inc., 1991.

[5] Tim Lindholm and Frank 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, MAY 2001, pp. 26-29.

[7] Jan Vitek, R. Nigel Horspool and Andreas Krall, “Efficient Type Inclusion Tests”, OOPSLA’97, Atlantas, GA, USA, October 1997, pp. 142-157.

[8] Andreas Krall, Jan Vitek and Nigel Horspool, “Near Optimal Hierarchical Encoding of Types”, ECOOP’97, Jyvaskyla, Finland, June 1997, pp. 128-145.

[9] Jeffrey Dean, David Grove, and Craig Cambers, “Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis”, ECOOP’95, Aarhus, Denmark, August 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, New York, 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, June/July 2001, pp. 17-23.

[13] C.C. Hwang, S.K. Huang, D.J. Chen, and David T.K. Chen,” Object-Oriented Program Behavior Analysis Based on Control Patterns”, APAQS 2001, Proceedings of the Second Asia-Pacific Conference on Quality Software, Hong Kong, December 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] Rakesh Agrawal and Ramakrishnan Srikant, “Mining Sequential Patterns”, Research Report RJ 9910, IBM Almaden Research Center, San Jose, California, October, 1994.

Appendix A

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

DigSim		
Types	**The number of event**	**Percentage**
Simple pattern	**70129**	**45%**
Sequence(S)	7826	5%
CL	34250	22%
RL2	28017	18%
RL3	36	0%
Compound pattern	**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 pattern	**13448**	**9%**
Total number of event	**156045**	**100%**

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

Fractal		
Types	**The number of event**	**Percentage**
Simple pattern	**19618**	**15%**
Sequence(S)	4700	4%
CL	14839	11%
RL2	61	0%
RL3	18	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**134158**	**100%**

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

Javac		
Types	**The number of event**	**Percentage**
Simple pattern	**101698**	**37%**
Sequence(S)	24181	9%
CL	44331	16%
RL2	20331	7%
RL3	12855	5%
Compound pattern	**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 pattern	**19062**	**7%**
Total number of event	**272193**	**100%**

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

Mapinfo		
Types	**The number of event**	**Percentage**
Simple pattern	**163696**	**53%**
Sequence(S)	36027	12%
CL	78906	26%
RL2	2736	1%
RL3	45027	15%
JP	1000	0%
Compound pattern	**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 pattern	**5740**	**2%**
Total number of event	**306904**	**100%**

***Table E** Control pattern distribution in % (MoleculeViewer)*

MoleculeViewer		
Types	**The number of event**	**Percentage**
Simple pattern	**159126**	**29%**
Sequence(S)	23704	4%
CL	135104	24%
RL2	300	0%
RL3	18	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**558202**	**100%**

***<u>Table F</u>* Control pattern distribution in % (ThreadX)**

ThreadX		
Types	**The number of event**	**Percentage**
Simple pattern	**36962**	50%
Sequence(S)	27621	37%
CL	9013	12%
RL2	220	0%
RL3	108	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**74449**	**100%**

***<u>Table G</u>* Control pattern distribution in % (Tic Tac Toe)**

TicTacToe		
Types	**The number of event**	**Percentage**
Simple pattern	**8200**	**20%**
Sequence(S)	2540	6%
CL	5263	13%
RL2	346	1%
RL3	51	0%

Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**40391**	**100%**

***Table H** Control pattern distribution in % (TrafficSim)*

TrafficSim		
Types	**The number of event**	**Percentage**
Simple pattern	**125295**	**22%**
Sequence(S)	46100	8%
CL	78953	14%
RL2	188	0%
RL3	54	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**563661**	**100%**

Table I ***Control pattern distribution in % (Tubes)***

Tubes		
Types	**The number of event**	**Percentage**
Simple pattern	**76530**	**13%**
Sequence(S)	8343	1%
CL	65350	11%
RL2	2741	0%
RL3	96	0%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**585210**	**100%**

Table J ***Control pattern distribution in % (WebDraw)***

WebDraw		
Types	**The number of event**	**Percentage**
Simple pattern	**79925**	**32%**
Sequence(S)	31321	13%
CL	43329	17%

RL2	850	0%
RL3	4425	2%
Compound pattern	**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 pattern	**0**	**0%**
Total number of event	**248353**	**100%**