A Trace Analysis Approach to Comprehend Features in Object-Oriented Effect Systems

Izuru Kume
Nara Institute of Science and Technology
Graduate School of Information Science
8916-5 Takayama, Ikoma,
Nara 630-0192, Japan
kume@is.naist.jp

Etsuya Shibayama
The University of Tokyo
Information Technology Center
2-11-16 Yayoi, Bunnkyo,
Tokyo 113-8658, Japan
etsuya@ecc.u-tokyo.ac.jp

February 27, 2009

Abstract

Many object-oriented systems, such as CASE tools manipulated by GUIs and Java Servlet applications responding to requests from internet clients, are driven by external inputs. We call such systems *object-oriented effect systems* where effects in runtime system states implement features correctly. Effects comprehension is thus an integral part in feature comprehension for object-oriented effect systems. However, effects comprehension in object-oriented effect systems is difficult and requires maintainers a new analysis method with different viewpoint for ordinary feature comprehension. In this paper, we propose a trace analysis method that is based on a trace model, and provides an effect analysis principle and a deductive object-oriented relational language as an analysis tool.

1 Introduction

An object-oriented interactive system is usually built on a calling framework [11] that hides low level event processing [7], and invokes application-side event handlers in response to users' inputs. Event handlers and the method invoked by them together implement an observable behavior in such a way that previously invoked methods produce effects (e.g. by instance variable assignment), and later invoked methods are influenced by the effects (by using the assigned value for a conditional branch, etc.) The influence of effects accumulates in a complex way because influenced methods themselves might influence other methods in a similar way.

We call such object-oriented systems with the above behavioral characteristic *object-oriented effect systems*. We call an observable behavior implemented by such accumulating influences of effects of an object-oriented effect system *a feature*, and we call influences among method invocations by effects *a coupling by effects* among the method invocations. Understanding a coupling by effects as a whole usually requires a time-consuming task to follow a data flow across method invocations.

We pursue a trace analysis approach in order to reduce maintainers' efforts to examine a coupling by effects without being worried by the problem of dynamic binding [13]. Comprehension of couplings by effects requires changes on trace analysis concepts with respect to trace models and analysis operations that affect the design of a trace analysis tool for feature comprehension in object-oriented effect systems. Figure 1 shows the hierarchy of trace analysis concepts required for comprehension of couplings by effects.

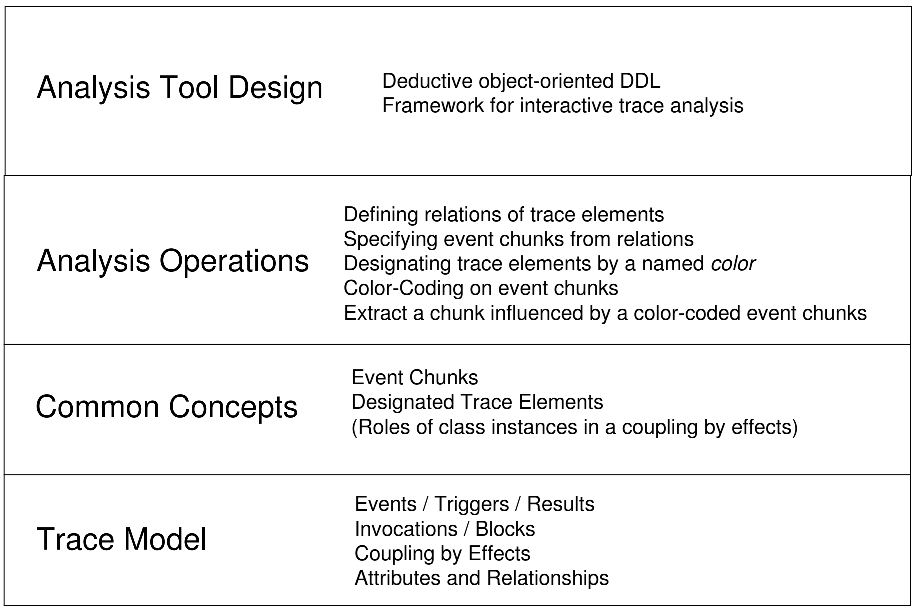
Analysis Tool Design
Deductive object-oriented DDL
Framework for interactive trace analysis
Analysis Operations
Defining relations of trace elements
Specifying event chunks from relations
Designating trace elements by a named *color*
Color-Coding on event chunks
Extract a chunk influenced by a color-coded event chunks
Common Concepts
Event Chunks
Designated Trace Elements
(Roles of class instances in a coupling by effects)
Trace Model
Events / Triggers / Results
Invocations / Blocks
Coupling by Effects
Attributes and Relationships

Figure 1: Conceptual Hierarchy for Effect Comprehension

In the rest of this report, we first show our motivating example in section 2. We explain our trace model in section 3, two analysis concepts to ingtegrate our approach to existing ones in section 4, analysis operations to comprehend a coupling by effects in section 5, and our deductive object-oriented data definition language in section 6, and an analysis result in section 7.

2 Motivating Example

2.1 Example RPG System

In this section, we explains an RPG (Role Playing Game) system that is implemented in Java and is used as an example object-oriented effect syste in this report. The the system has about 1,000 lines of the source code. The framework classes of the system are shown in Figure 2. The class diagram at the left side of the figure shows the associations just after the system initialization. Class `RPGCharacter` represents RPG characters in a game. Class `RPGCommand` represents applicable actions of RPG characters. All information about a RPG character is shown as primitive number values and strings that are presented on the corresponding GUI panel,[1] an instance of `CharacterBrowser`. The GUI panel also provides the player of the RPG character with the interface to select actions in a battle with another RPG character. Class `BattleField` represents a battle field where two RPG characters attack each other. RPG characters that share the same `BattleField` instance can participate in the same battle.

A game player can select an attack action (an instance of `RPGCommand`) and an attack target (an instance of `RPGCharacter`) by operating the GUI panel. The selected action and target are passed to a shared battle field (a `BattleField` instance) as arguments of a method. Bing the method invoked, the battle field checks if it is the first time for the method to be invoked. If it is the first time, the battle field creates an instance of `BattleProcessor`, assigns the created instance to its instance variable, and sets up object references shown as the associations at the right side of Figure 2. The assignment of the created instance into its instance variable indicates that the method has been invoked once. At the second invocation, the battle field does the same thing, and after the set up it invokes a method on the two `BattleProcessor` instances in order to process the total battle results.

A class `BattleProcessor` is responsible for processing the result of an attack by an RPG character to another. It checks the success or failure of the attack based on the attacker's skill (obtained from the attack command), and the defender's skill, and reduces the defender's HPs (hit points) if the attack is successful. The `RPGCharacter` instances (each passed as a target of the attacks) reflects the changes of their HPs in the displayed numerical values and string messages.

A sequence diagram in Figure 3 shows the process where an RPG user makes two RPG characters attack each other just after the syetem initialization in terms of objects collaborations. Only the Main class and two attack commands (`command_1` and `command_2`), and their common battle field (`battle_field`) are

[1]The system provides no 3D presentation at all.

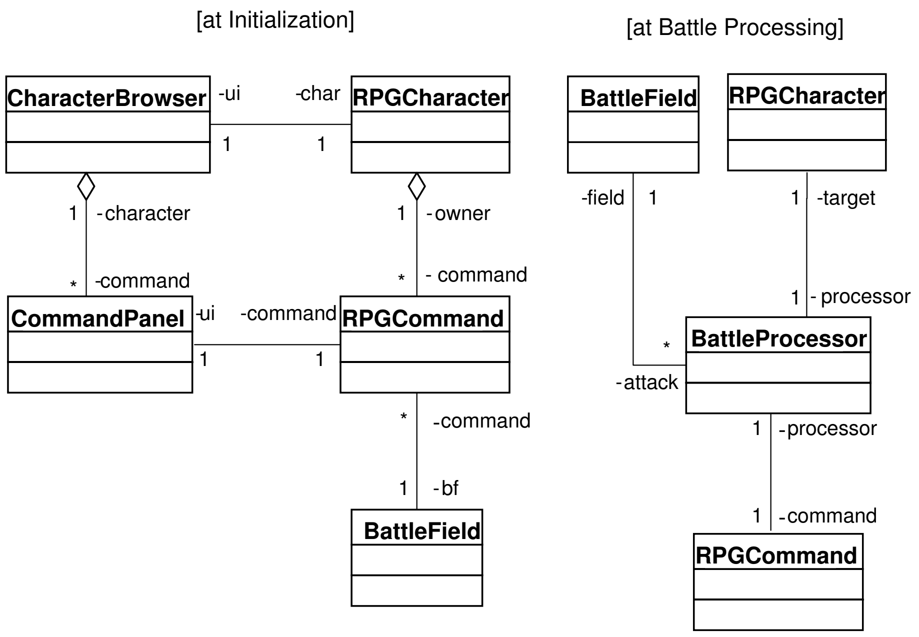

Figure 2: Framework Classes

shown although more objects participate in the process. Balloons in the diagram contains occurring events in the process such as the creation of class instances, a conditional branch by the battle field, and the effects produced by assignments to instance variables which accompany the various system state setups in the process.

Notice that the method invocations are separated under three mutually independent invocation trees, which are triggers by the Main method and the two inputs by the user. Here we say two method invocations are ‘independent’ if none of them doesn’t invoke another directly nor indirectly. However, several method invocations in the initialization and the first input handling process influences the later ones by their effects, that is, the method invocations (and their containing invocations trees) are coupled by effects. It is important that the couplings by effects are implemented as the data flows across the method invocations but along the events in the balloons.

Let’s see several steps that contributes to the data flows. The attack commands at their creation set up a path to their common battle field, and the attack commands at their later method invocation (triggered by two inputs `select command_1` and `select command_2`) obtained the battle field by referencing the path. As a result, their later method invocations are affected by their setups at the creation time, in other words the methods are coupled by

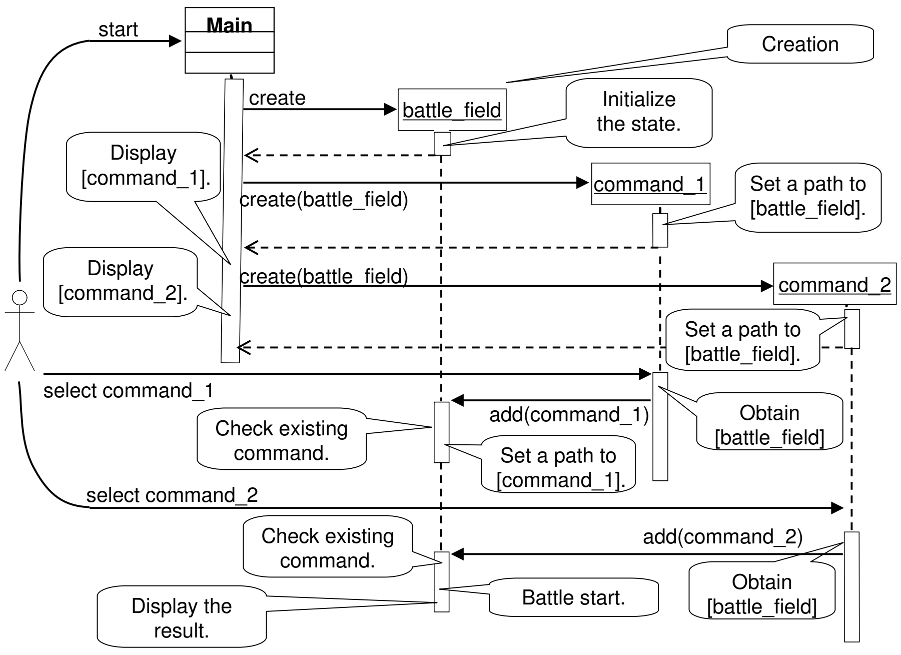

Figure 3: Object Interactions

the effects of the creation. We can find many such examples couplings by effects in the whole process of an object-oriented effect system.

2.2 Assumed Trace Analysis Operations

It is inevitable for maintainers to understand the roles of classes and particular class instances on understanding the implementation of the coupling by effects in the above battle processing feature. What kind of supports do the maintainers require for such a program comprehension purpose? A trace analysis approach is desirable in order to avoid the problem of dynamic bindings[13], which is a famous obstacle specific to comprehension of object-oriented programs.

The maintainers often need a support for iterative collaboration recovery, an approach proposed by Richiner and Ducasse [9] for example, because the examined coupling by effects accompanies the object collaborations in Figure 3. When the maintainers encounter `battle_field`, which is one of the key instances for the feature comprehension, they likely try to the execution history related to `battle_field`. In the examination, the maintainers possibly need to know "Where the value of the instance variable was assigned?", for which examination the Omniscient debugger [3] provides the maintainer with a good support.

In addition to the above examination from a fine grained viewpoint, the maintainers possibly require a coarse grained examination during their comprehension process. Assume that the maintainers try to locate where the calculation of the battle results actually starts in the whole execution. The battle processing actually starts just after the check in the method of `battle_field` whether it is the first method invocation or not. Thus the maintainers accomplish the comprehension subtask if they can efficiently locate the second method invocation on `battle_field`. If the maintainers lack of the implementation knowledge about `BattleField`, they must find the trace location that are specified as "A location in the second input handling processing which are influenced by an effect of the first input handling process".

Although several approaches [10, 4, 5] support dependency analysis caused by effects of object creations and aliases, nothing provides a direct support for understanding the roles of classes or particular class instances in the implementation of a coupling by effects, as well as we know. A novel trace analysis method that allows maintainers to examine a coupling by effects both from a fine-grained viewpoint and a coarse-grained viewpoint, and provides a direct support to comprehend the roles of classes and particular class instances in the implementation of the coupling by effects is desired.

3 Trace Model

Our trace model represents a program execution in terms of *events*. Modeling a program execution in terms of events enables the fine-gtained viewpoint in section 2.2, and suits the bottom-up comprehension stratecy [8].

Each event is triggered by an operation including: (1) constant value introduction, (2) primitive value calculation, (3) instance and array creation, (4) value duplication (assignment to a local variable), (5) getting a value of a class variable, an instance variable, and an array element, (6) calling a method, (7) returning from a method, (8) conditional branch, (9) assignment to a class variable, an instance variable, and an array element. Executing an operation in type (9), an assignment to an instance variable for example, results in generation of an effect. Executing an operation in type (6), (7), or (8) results in a change of control. An operation in the rest of the types results in creating or obtaining values that are categorized into object reference, array reference, and primitive values. Notice that the result of such an operation is not the value itself but the creation of the value. Events together with their triggering operations and their results are represented as trace elements and thus they are query subjects.

An operation, say op_β depends on another operation, say op_α if one of the following conditions is satisfied: (1) op_β uses the result of op_α, (2) op_α is a conditional branch and op_β is executed as a result of the conditional branch, (3) op_α is an assignment of a value to a class variable, an instance variable, or an array element, and op_β gets the assigned value. An event, say ev_α, influences another event, say ev_β if and only if the triggering operation of ev_β depends on the triggering operation of ev_α. Our trace model represents the dependency

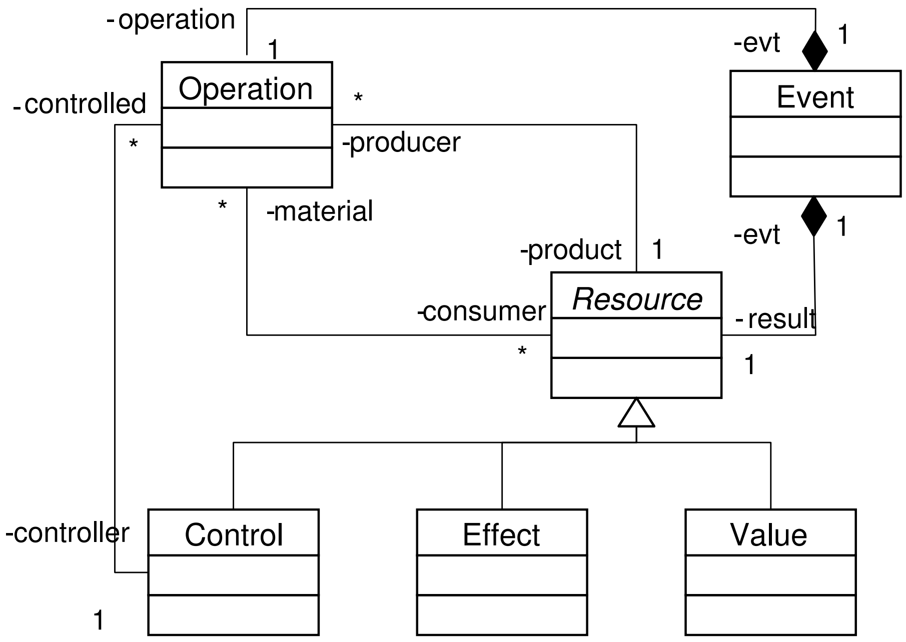

Figure 4: Event and its Constructs

among operations and thus implies the influence among events. The transitive closure of influences starting from an event e forms the accumulated influences of e.

Let's consider the events that occur in the coupling by effects in our example system. The main method invocation contains two events ev_c and ev_{bf} that represent the creation of a battle command, and `battleField` respectively. The first event handling process contains an event ev_a that represents an assignment of the battle command to the instance variable of `battleField`. The second event handling process contains two events ev_{get} and ev_{cond} that represent an operation to get the assigned value and the second conditional branch by `battleField` respectively. From the above definition, ev_c and ev_{bf} influence ev_a directly, and influence ev_{cond} indirectly through ev_a and ev_{get}. Thus the influence of ev_c and ev_{bf} accumulate. (The concept of accumulation of influences in a method is similar to program slice [12, 1], although we have no slicing criterion and take account of runtime concepts including effects.)

Our trace model contains as trace elements *method invocation* and *blocks* of control flows in order to show where events occur.[2] A method invocation starts with a method call event and ends with a return event. A method invocation

[2]In our current implementation, executed operations has an attribute to show the line number in a source code.

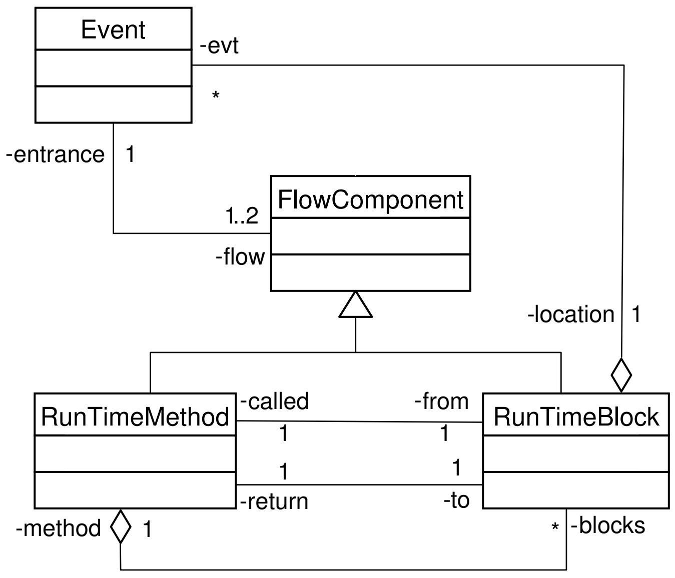

Figure 5: Method Invocation and Block

is decomposed by blocks. A block contains a sequence of events in the order of execution. A block starts when the current method starts, when a called method returns, or when a conditional branch starts. A block ends when the current method returns, a method is called, and a conditional branch ends. A conditional branch occupies a single block alone.

Trace elements have attributes whose value can be used to express query conditions. For example, an instance variable assignment has the variable name as its attribute. Some attributes have trace elements as their value. The event ev_a in the above example has as its attribute value the result of ev_c that represents the creation of the assigned value. (Recall that the result of instance creation is not the created instance but the creation itself.) Usually the attributes of trace elements are used for queries to specify areas of concern and to arrange the method invocations to form a locating result. On the other hand, influences among events are used to obtain coupling points.

Class diagram 4 illustrates the class of events and their constructs. Class `Event` represents events. Two abstract classes `Operation` and `Resource` represent triggering operations of events and their results respectively. `Value`, `Control`, and `Effect`, the subclasses of `Resource`, classify the results into value creation, effect generation, and change of control respectively. Association `producer-product` relates an operation with its result, while `consumer-material` represents that an operation depends on a result of another operation. The influence by a conditional branch is distinguished by `controlled-controller`.

Classes `RunTimeMethod` and `RunTimeBlock` in class diagram 5 represent invoked methods and blocks respectively. An abstract class `FLowComponent` abstracts them. The association `evt-location` represents events belonging to a block, while `blocks-method` represents decomposing of an invoked method by blocks. Calling a method and returning to a method is represented by `called-from` and `return-to` respectively. Association `flow-entrance` represents an event to start a method invocation or a block.

4 Common Concepts

In section 2.2 we addressed a situation that maintainers try to find a class instance (`battle_field`), which is a key to understand the role of classes or class instances in a coupling of effects, using the influence relationship between among the events in the two event sets each of which represents an input handling process. Notice the important fact that the maintainers effort is based on a idea which is completely different from those of existing approaches such as trace matching [6, 2] and iterative collaboration recovery [9] that rely on known program elements. Instead of known program elements such as class names or method names, the maintainers rely on the selection of the two event sets correspondence to the inputs, and the inherent influence relationship among events. The combination of events selections and the influence relationship works well in this case.

Let's consider a set of trace elements from a difference viewpoint. On the one hand, our trace model enables maintainers with a view of program execution in terms of fine-grained trace elements. On the other hand, the fine-grained trace elements and their relationships increase the amount of information and the complexity of the whole trace. It is natural for maintainers to divide the whole trace into several subsets, or to select a set of trace elements in which they believe to find a key to comprehend the roles of classes and class instances in a coupling by effects.

We call a set of a single type of trace elements which are selected according to maintainers' current concerns and understanding *a trace chunk*. We call a trace chunk containing events *an event chunk*. From our experience, maintainers often find some trace elements important for program comprehension as we saw in section 2.2 that the conditional branch derives `battle_field` from which we can well guess class `RPGCommand` implements attack commands. We call such trace elements that are thought important subjectively by maintainers `designated trace elements`.

We think finding designated trace elements as one of the important subgoal in a trace analysis process, and setting trace chunks as a means of finding designated trace elements. We believe that many analysis operations to comprehend a coupling by effects can be defined in terms of setting trace chunks and finding designated trace elements from the chunks. On our definition, we can introduce reusable patterns for suitable trace chunk settings and automate some processes to select designated trace elements. An analysis tool that supports the patterns and the automation can make a trace analysis task efficient. Operations of several existing trace analysis approaches can be defined in terms of trace chunks and designated trace elements. In this way, we can integrate our trace analysis approach to comprehend coupling by effects with existing ones. Such integration is important to design a trace analysis tool that supports various kinds of analysis operations as we saw in section 2.2.

5 Analysis Operations to Comprehend Coupling by Effects

5.1 Relational Operations

Our trace analysis approach adopts an object-relational data model to manages trace elements. Maintainers can directly access trace elements by displaying a relation that contains the trace elements as a table. There are two kids of tables. *An attribute relation* represents a single type of trace elements with their attributes. In this case, the attributes of the trace elements coincides with the attributes of the relation. For example, `RunTimeInvocation` instance are contained in a relation with the same name accompanied by their class names, method names, method signatures, and the `Value` instances as their method receivers and arguments.

Attribute relations represent the contents of the whole trace. All attribute relations except for one can't be redefined nor be modified by maintainers. Maintainers can recursively define new relations among trace elements by a deductive object-oriented data definition language, which we will explain in section 6.

Any trace chunk can be obtained as the value set of an attribute[3] of some relation. Currently it is the only way to obtain trace chunks.

5.2 Trace Element Designation

In our approach trace elements is interactively designation through a table that displays the contents of some relation. There are two styles of operation for trace element designation. In any way, a named element *color* is created and is associated with the designated trace element. Each color has a unique name, and thus maintainers can explicitly specify any designated trace elements by the name of their associated colors.

Colors have their names and the associated trace elements as their attribute values. Any color exists if and only if it is created by maintainers, and the attribute relation for colors increases as a new color is created.

Maintainers can directly select a trace element from a displayed table and give a name to the created color. As for another way of trace element designation, several method invocations can be designated as a result of root-selection operations explained later in section 5.4.

[3]Not an attribute of trace elements but an attribute of a relation.

5.3 Color-Coding

Colors can be used not only for trace element designation but also for making a set of events as an influence source on later events. Our approach defines an operation `color-coding` on an event chunk. All events that are directly or indirectly influenced by some event in a color-coded event chunk is associated to the color. Color-coding operations are often used to select an event chunk that are influenced by another event chunk.

5.4 Invocation Root Selection

Our approach defines `root-selection` operation on method invocation chunks. Given a method invocation chunk, the operation first forms the smallest method invocation trees to cover the chunk, and then selects the root invocations of the covering trees. The selected root invocations are designated by automatically created colors with a series of unique names according to some systematic naming rule.

Root-selection operations are used so that maintainers can locate starting points in a source code from which all method invocations in the given chunk can be followed, for example.

6 Data Definition Language

Our approach provides maintainers with a deductive language interface to recursively define a new relation of trace elements by a set of logical clauses in the following form:

```
<head> :- <body-1>, ..., <body-n>.
```

The defined predicate `<head>` is expressed as $p(t_1, \ldots, t_k)$, where the predicate symbol p is a sequence of English letters and digits starting with a lower-case English letter. Predicates `<body-i>` in the body have the same form of the defined predicate, or a system predicate. System predicates are categorized into *attribute predicate* that expresses trace elements and their attributes, *color predicate* that expresses colors, and *twisting predicate* that expresses coupling points influenced by color-coded events. Color predicates and twisting predicates reflect the results of color-coding operations in sectino 5.3, while attribute expresses the trace elements themselves. All predicate arguments must be a variable, a numerical number, and a string. All trace elements are expressed in a variable.

An attribute predicate expresses a trace element and its attributes. Below, we show an attribute predicate to express an operation to get the value of a class variable or instance variable. The predicate name `CC_FieldGet` represents the class name of the operation. The variable `Op` expresses the operation itself. The predicate arguments express its attributes: the name of the class to declare the variable, the result of an operation to get the instance to have the variable or null (in the case of class variable), the variable name, the variable signature, and the result of assignment operation that assigned the current value of the variable.

```
CC_FieldGet[Op](class, instance,
      field_name, field_type, effect).
```

Because `CC_FieldGet` is a subclass of `Operation` (see class diagram 4), any of its instances inherits the attributes of `Operation`. The attribute predicate below expresses the attribute, that is, operation type, byte code number, occurring block, and the line number in the source code.

```
Operation[Op](type, opcode, block, line)
```

The predicate color predicate below (`#Color`) below expresses that a color `C` has a name `name` and is associated with `element`. The twisting predicate below (`#ColorTwisted`) expresses that a coupling point `event` is influenced by an event color-coded by `C`. A color predicate and a twisting predicate are often combined to select coupling points.

```
    #Color[C](name, element)
    #ColorTwisted[C](event)
```

7 Query Process in Practice

Now we will see how we actually accomplished the query process addressed in section 2.2 by applying a prototype analysis tool that implements the analysis operations in section 5 and the data definition language. As a result, we finally obtained a method invocation on `battle_field` as a result. We first input the following clause to define `actionMethod` to select the methods invocations from the Swing framework. We use the knowledge about Swing API that event handlers are named 'actionPerformed' and have a signature "(Ljava/awt/event/ActionEvent;)V"[4].

```
actionMethod(Mtd)
  :- RunTimeInvocation[Mtd](
  _, "actionPerformed",
      "(Ljava/awt/event/ActionEvent;)V").
```

We displayed the relation `actionPerformed` and saw that two methods were invoked by the Swing framework. We examined the serial number of the invocations that represents their invocation order, and associated a color named `act_1` to the first invocation, and a color `act_2` to the second invocation. Next, we defined `select1` and `select2` to select the two sets of events in the processes started by `act_1` and `act_2`, respectively. The color predicates are used to distinguish the processes, and the set of events are selected by mutual visit of method invocations and blocks. We specified our areas of concern by color-coding the selected sets by their corresponding colors.

[4]It is a signature format for Java VM.

```
call(Mtd, ColorCode) :-
  #Color[ColorCode](_,Mtd), RunTimeInvocation[Mtd](_, _, _).

call(Mtd, ColorCode) :-
  ccBlock(Blk, ColorCode), Event[_](Ctrl, Ivk, Blk),
  Invoke[Ivk](_, _, _, _), FlowComponent[Mtd](_, Ctrl, _, _),
  RunTimeInvocation[Mtd](_, _, _).

ccBlock(Blk, ColorCode) :- call(Mtd, ColorCode),
  FlowComponent[Blk](Mtd, _, _, _), RunTimeBlock[Blk](_, _, _).

colorCoded(Evt, ColorCode) :-
  Event[Evt](_, _, Blk), ccBlock(Blk, ColorCode).

select1(Evt) :-
  colorCoded(Evt, ColorCode), #Color[ColorCode]("act_1", _).

select2(Evt) :-
  colorCoded(Evt, ColorCode), #Color[ColorCode]("act_2", _).
```

Next, we defined a predicate `coupled` to select the set of coupling points that are in `act_2` and are color-coded by `act_1`. Notice that a color twisting predicate and a color predicate are combined to select the coupling points. We also defined `coupledMethod` to select invoked methods that directly execute the coupling points.

```
coupled(Evt) :- select2(Evt),
  #ColorTwisted[Act1](Evt), #Color[Act1]("act_1", _).

coupledMethod(Mtd) :- coupled(Evt),
  Event[Evt](_, _, Blk), RunTimeBlock[Blk](_, _, _),
  FlowComponent[Blk](Mtd, _, _, _).
```

Last, we displayed `coupledMethod`, and selected the set of invocation roots from the displayed column. In this case, the selected set only contains the method invocation on `battle_field`, which is associated with a color named `"Root<selectedMethod@0>_0"`. We defined `printSelection` to examine the class (`C`), the method name (`M`), and the method signature (`S`).

```
printSelection(C, M, S) :-
  #Color[_](
    "Root<selectedMethod@0>_0",Mtd),
  RunTimeInvocation[Mtd](C, M, S).
```

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Conference on Programming language design and implementation*, pages

246–256. ACM, 1991.

[2] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA*, pages 385–402. ACM, 2005.

[3] Bil Lewis. Debugging backwards in time. In *International Workshop on Automated Debugging (AADEBUG)*, 2003.

[4] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Exposing side effects in execution traces. In *International Workshop on Program Comprehension through Dynamic Analysis*, pages 11–17, 2007.

[5] Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *International Conference on Program Comprehension*, pages 59–68. IEEE, 2007.

[6] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA*, pages 365–383. ACM, 2005.

[7] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, 1990.

[8] Srinivas Neginhal and Suraj Kothari. Event views and graph reductions for understanding system level C code. In *International Conference on Software Maintenance*, pages 279–288. IEEE, 2006.

[9] Tamar Richner and Stéphane Ducasse. Using dynamic information for the iterative recovery of collaborations and role. In *International Conference on Software Maintenance*, pages 34–43. IEEE, 2002.

[10] Maher Salah, Trip Denton, Spiros Mancoridis, Ali Shokoufandeh, and Filippos I. Vokolos. *Scenariographer*: A tool for reverse engineering class usage scenarios from method invocation sequences. In *International Conference on Software Maintenance*, pages 155–164. IEEE, 2005.

[11] Steve Sparks, Kevin Benner, and Chris Faris. Managing object-oriented framework reuse. *IEEE Computer*, 29(9):52–61, 1996.

[12] Mark Weiser. Program slicing. In *International Conference on Software Engineering*, pages 439–449. IEEE, 1981.

[13] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.