Specification and Runtime Verification of API Constraints on Interacting Objects

Fang Deng, Haiwen Liu, Jin Shao, Qianxiang Wang
School of Electronics Engineering and Computer Science, Peking University
Beijing, China
{dengfang07, liuhw08, shaojin07, wqx}@sei.pku.edu.cn

Abstract—Most applications need to invoke some Application Programming Interfaces (APIs), e.g. JDK (Java Development Kit) API. When invoking those APIs, applications must follow some API constraints. Violation of these constraints will lead to some severe program defects. To detect this kind of defects, lots of static and dynamic approaches are explored, using formally described API constraints. While most existing approaches explore API constraints on a single object, this paper focuses on API constraints on interacting objects (COIOs). We proposed a novel specification language LACOIO (Language for API Constraint on Interacting Objects) which can cover all state-of-the-art reported API COIOs. We implemented a LACOIO compiler which can automatically generate monitoring code for runtime verification from API constraints that are specified using LACOIO and implemented the runtime verification framework. We evaluated the proposed approach by comparing it with Tracematch – a popular constraint specification language.

Keywords—Runtime verification; runtime monitoring; specification; program analysis.

I. INTRODUCTION

Most applications need to invoke some Application Programming Interfaces (APIs), e.g. JDK (Java Development Kit) API. When invoking APIs, applications must follow some API constraints. For example, one constraint related to JDK is: “instance of java.io.InputStream must be closed explicitly”. Violation of these constraints will lead to some severe program defects. To detect this kind of defects, lots of static and dynamic approaches are explored [9, 12]. For both kinds of approaches, formally described constraints are widely used in the detecting process. Static analysis tools can read these formally described constraints and check the code automatically, while dynamic verification frameworks can use these constraints to generate monitors automatically to verify applications at runtime. Finite state automaton (FSA), temporal logic, and regular expression are typical formal methods used to describe API constraints [3, 5, 8].

While most existing approaches explore API constraints on a single object [7, 8, 11], some studies explore API constraints on multiple interacting objects (COIOs) [2, 10]. For example, one constraint related to JDK (we call it SafeIterator Constraint) is: “an array list of java.util.ArrayList cannot be updated when it is being traversed by an iterator of java.util.Iterator”. Fig. 1 shows the FSA-based expression for this constraint [3].

However, the event symbols in this FSA reveal only the event names without information about which objects these events belong to. Actually, this FSA makes some implicit assumptions: event CREATE_LIST occurs on a java.util.ArrayList object al; event CREATE_ITERATOR occurs on the java.util.Iterator object i which is created by al; all UPDATE events belong to the same object al; and all NEXT events belong to the same object i.

When an API constraint involves a group of objects like this, it is very difficult to distinguish which object an event is associated with. Some approaches [1, 5] add free variables to events (symbols in the FSA) to address this problem. These approaches use a variable binding strategy to associate objects with events. In Fig. 2, two free variables al and i are introduced as well as two operators “=” and “.”. Edge “al=CREATE_LIST” indicates that the returned object of the operation which is matched to the event CREATE_LIST is bound to object al. Edge “al.UPDATE” matches the updating operations which happen on object al. Other edges can be explained similarly. Specification in Fig. 2 has two benefits. First, it points out which object an event is associated with; secondly, it points out how objects al and i are bound together, i.e., i is the Iterator object that is created by al.

However, for constraints on interacting objects whose number is not stable, it is still very difficult to describe them formally, even with existing free-variable-based approaches (See section II for details). Moreover, the approach of formal specification has great impact on the implementation of corresponding verification framework. In this paper, we proposed a novel specification language LACOIO (Language for API Constraint on Interacting Objects) to describe API constraints. The proposed language is powerful enough to describe state-of-the-art reported API COIOs easily and leads to efficient runtime verification.

The rest of the paper is organized as follow: In Section II we present a COIO example to introduce our motivation and give a brief introduction to COIO specification problem. In Section III we present the proposed specification language LACOIO. In Section IV we introduce the implementation of LACOIO compiler and the runtime verification framework. Section V is the evaluation of our work. Section VI is the introduction of related work. We conclude our paper in Section VII.
Figure 1. FSA for the constraint on ArrayList and Iterator.

Figure 2. Extended FSA for constraint on ArrayList and Iterator.

Figure 3. A piece of sample code that violates SafeJDBC constraint.

Figure 4. An example which tries to use FSA with free variables to specify SafeJDBC.

II. PROBLEM STATEMENT

To clarify the complexity of API COIO verification, we present another motivating example. The constraint is about JDBC (we call it SafeJDBC Constraint): "if two java.sql.ResultSet objects (e.g. rs1 and rs2) are created by the same java.sql.Statement object (e.g. stmt) through Statement.executeQuery(...) method, the first ResultSet rs1 becomes unavailable after rs2 is created" [4]. In other words, we cannot access rs1 through any of its methods after rs2 is created. Fig. 3 is a piece of sample code that violates SafeJDBC. SafeJDBC involves at least 3 objects, stmt of class Statement, the rs1 and rs2 of class ResultSet. Now we try to describe it by FSA which introduces free variables similarly as

Fig. 2. In Fig. 4, the real line elements describe the violation pattern involving three objects: stmt, rs1, rs2. “-1” in the FSA stands for the state when a violation is detected. Usually, at runtime, such an FSA instance will be created to verify behavior of the three objects.

However, the problem is: what if a third instance of ResultSet rs3 is created? Even if we can extend the FSA to describe the violation pattern when rs3 is involved (see the dotted line elements in Fig. 4), what if more instances of ResultSet are created?

There are two facts that make it problematic to describe the constraint by FSA, even with introduction of free variables. (1) The verification must be carried out on instance-level (not the class-level). Although this type of constraint involves finite classes, the number of involved objects is undetermined. It is impossible to enumerate all runtime instances in an FSA. (2) The other characteristic of such kind of constraints is that, whether behavior of one object will lead to violation of the constraint depends on behavior of another object of the same class. This situation prevents us from describing the behavior of such two objects in two separate FSAs.

A. Problem-related Terminologies

Before introducing the details of COIO specification problem, we first explain some relevant terminologies.

Monitor. A monitor is an instance for a constraint. It inspects and verifies the behavior of a set of interacting objects. In Fig. 4 the monitor is an FSA instance which is created to verify the behavior of stmt, rs1, rs2, rs3.

Monitee. At runtime if an object is involved in the verification and will be monitored, we call it a monitee. In Figure 4, object stmt, rs1, rs2, rs3 are monitees.

Monitee set. All monitees that are verified in one monitor constitute the monitee set of the monitor. In Fig. 4 the monitee set is {stmt, rs1, rs2, rs3}. The monitee set of a monitor can be divided into several subsets according to the monitees’ types. The monitees of a certain class C constitute the monitee set of C. For example, there is a monitee set {rs1, rs2, rs3} of ResultSet.

Binding point. Assume C1, C2 are two classes, C1 has a method m whose signature involves C2, we call m a binding point for C1 and C2, e.g., method ResultSet.executeQuery(String str) is a binding point for Statement and ResultSet.

B. Problem of Specification

In this paper we restrict our focus on COIO whose monitee set is defined as \{k\} \cup \{i | i BP i\}, where k is an instance of a
class C. k is called the **core object** in the monitor while C is called the **core class** of the COIO. We reviewed state-of-the-art literatures and found even with this restriction the proposed approach can cover all state-of-the-art API constraints.

From the definitions of monitee set of COIO monitor, we know that there is always one core object in the monitee set of each COIO monitor and all the other objects are involved by binding relationship. This indicates a one-to-many mapping between the object of core class and objects of other involved classes in COIO. Constraint SafeJDBC is such a COIO: `stmt` is the core object in its monitee set and all the ResultSet typed objects that are created by `stmt` are involved in the same monitor by binding point “ResultSet
`stmt`.executeQuery (String str)”. Fig. 5 illustrates the monitee set, its monitees, and related objects.

The key challenge for traditional FSA monitor is that its monitee set must be a fixed set. That means, for FSA, we must present each possible object of the monitee set in FSA description, which is impossible in the aforementioned case before runtime because the number of objects in the monitee set is undetermined. For example, in Fig. 4, we must include all the objects `stmt`, `rs1`, `rs2`, `rs3` to describe the constraint. It is likely that more ResultSet objects are created in the near future. So the FSA-based approach can’t describe such undetermined cases.

### III. LANGUAGE FOR API CONSTRAINT ON INTERACTING OBJECTS

LACOIO can describe complex COIOs in a succinct way and make specifications easy to understand. Complete grammar and more examples can be found in [13]. In this section, we explain the LACOIO specification with the example of SafeJDBC constraint (see Fig. 6).

#### A. Monitor

A monitor is defined with the keyword **monitor**. At runtime a monitor instance represents an instance of COIO verification case.

#### B. Monitee set management and binding

The monitee set is defined with keyword **monitee**, contains the objects that are involved in a monitor instance. The monitor usually contains two kinds of monitee sets: monitee set for core class which contains only one core object and monitee sets for other classes. In our specification, we do not distinguish these two kinds of monitee sets. Monitee sets are maintained dynamically at runtime and do not need to be determined before runtime. In the example, two monitee sets `@stmt` and `@rs` are specified respectively for class Statement and ResultSet. At runtime each SafeJDBC monitor maintains such two sets. Since Statement is the core class for SafeJDBC, there is only one object of Statement in `@stmt` which is the core object. Whenever a ResultSet object is obtained by the binding point “ResultSet
`stmt`.executeQuery (…)”, the returned ResultSet object is bound to the core object and thus put into `@rs`.

#### C. User-defined variable

We allow users to define their own variables with keyword **var** to store historical states that can be acquired in the future, e.g. a reference to certain monitee in the monitee set, a status tag, etc. In Fig. 6, **var** ResultSet `$last` defines a variable called $last of type ResultSet. It is used to store the reference to the most recently created ResultSet object.

#### D. Event

When a method is invoked on a monitee, it will be mapped to the corresponding event that is specified in the constraint according to particular mapping rules. It is similar with the process of mapping method calls to edges in an FSA. An event declaration consists of event type, event name, contract part, pre/post-condition, revise part, and action.

**The contract part.** The contract part specifies the rules that how method invocations should be mapped to events. For example, in Fig. 6, the contract part “@stmt.executeQuery (String)” of event STQUERY states that invocation to “executeQuery(…)” methods on the object which is in the `@stmt` set should be mapped to the STQUERY event.

**Pre/Post-condition.** Pre/Post-conditions are defined for every event in the constraint, which are identified by “{ }” before/after a contract part. They are part of the mapping rules. Only when pre/post-conditions are both true can a method invocation be matched to an event successfully. They are boolean expressions on $Target, $args[n], $return and the user-defined variables. $Target, $args[n], and $return are reserved keyword that are used to refer to the target object, the n-th argument, and the return value of a method call, respectively. The pre/post-conditions are checked before/after the invocation of the method which is specified in an event declaration. When
The structure of the runtime verification framework is shown in Fig. 7. JVM objects pool is presented in the left side in which there are objects of verification concern. The squares in the objects represent methods that have been instrumented with probes. The probes send messages to the corresponding objects. The handler of the envoy will process the messages sent from the probes. And if necessary, the envoy will pass the message to the monitor for further processing.

IV. THE IMPLEMENTATION

The specification file will be compiled to monitoring code. The compiler takes LACOIO specification as input and compiles it into two parts. (1) One is an AspectJ [6] file which acts as a middle representation file. In general, the responsibility of the AspectJ file is to declare probes that map method invocations to events and transfer messages at runtime. Pointcuts will be generated according to event contract declarations. The code in an advice will be generated to handle constraint violations. For example, in the SafeJDBC specification example, when “next(...)” is invoked on one of the ResultSet monitees and ResultSet object referred by Starget does not equal Slast, validate event RSNEXT is matched and constraint violation is reported.

The structure of the runtime verification framework is shown in Fig. 7. JVM objects pool is presented in the left side in which there are objects of verification concern. The squares
So with our approach we only need to check whether "get2" is not necessary for identifying a violation. All that is necessary to know is that the three events did happen before "selected2" happens.

Although LACOIO does not express the logic formula explicitly, by introducing user-defined variable properly it can express all the constraints that can be expressed by FSM and regular expression. Our motivation to hide the logic formula is that we find many state-of-the-art constraints are not pure temporal problem and can be expressed by our language more straightforwardly. (See [13] for more specification examples).

B. Performance

In this sub-section, we illustrate the performance improvement that our language design brings to the verification of some typical type of complex COIOs like SafeDropDownList (mentioned in section V.A) which involve interacting objects of the same class. To evidence this point, we verify a Java application which manipulates one DropDownList and multiple ListItems with respect to the SafeDropDownList constraint. The experiments are conducted on a PC equipped with Intel Pentium IV 3.0G Hz processor, a 2GB RAM, and a single 300GB 7200 RPM SATA disk. The operation system is Ubuntu 9.10. JDK version is 6.0.

Fig. 12 shows the execution time when Tracematch and our approach are applied. We run the application multiple times. X-axis shows the total number of ListItems that are created in a particular round of execution. Y-axis shows the time cost for the execution. As we expect, as the number of ListItems increases, the execution time increases for both Tracematch and our approach. But the increasing rate for Tracematch is approximately 7.4 times as that for our approach.

The main reason for this result is that, in LACOIO, monitee sets are declared explicitly and maintained at runtime. Since the DropDownList object is the core object of the monitor, the behavior of all ListItems that belong to the same DropDownList will be verified in one monitor instance. Compared to LACOIO monitor, the Tracematch monitor specified in Fig. 11 states the behavior pattern of three objects: DropDownList l, ListItem i1 and i2. In fact, the specification implies that information of each pair of ListItems which belongs to the same DropDownList are tracked and matched in one pattern. If there are three ListItem i1, i2, i3, the behavior of object pairs <i1,i2>,<i1,i3> and <i2,i3> will be verified by

```java
1 monitor SafeDropDownList{
2 monitee @list<DropDownList>,@items<ListItem>;
3 var ListItem $selected = null;
4
5 init DropDownList.new()
6 { @list.add($return) }
7 { /* empty action */}
8 event FINDITEM
9 { @list.findItemByName(String)
10 { @items.add($return) }
11 { /* empty action */}
12 event SELECT
13 [args[0]==true && $selected==null]
14 @items.setSelected(boolean)
15 { $selected= @target; } 
16 validate SELECT_FALSE
17 [args[0]==false && $selected==$target]
18 @items.setSelected(boolean)
19 (System.out.println("Error Reported"); }
20 event DESELECT
21 [args[0]==false && $selected==$target]
22 @items.setSelected(boolean)
23 { $selected = null; }
24}
```

Figure 10. SafeDropDownList Specification in LACOIO

```
tracematch(DropDownList l, ListItem i1, ListItem i2) {
  sym get1 after returning(i1) :
    call(* DropDownList.findItemByName(..)) && target(i1);
  sym get2 after returning(i2) :
    call(* DropDownList.findItemByName(..)) && target(i1);
  sym select after : target(i1) && private(boolean b) ( 
    call(void ListItem.setSelected(boolean) && args(b) && if(b));
  sym deselect after : target(i1) && private(boolean b) ( 
    call(void ListItem.setSelected(boolean) && args(b) && if(!b));
  sym select2 after : target(i1) && private(boolean b) ( 
    call(void ListItem.setSelected(boolean) && args(b) && if(!b));
  distinct i1, i2,
  (get1 get2* select (get1|get2|select)* select2) | (get1 select (get1|select)* get2 (get1|get2|select)* select2) 
  { error("Selecting " + i2 + " without deselecting " + i1); }
}
```

Figure 11. SafeDropDownList Specification in Tracematch

SafeDropDownList states that: “multiple ListItems in a DropDownList cannot be set to 'selected' at the same time”. In other words, before trying to select a ListItem by calling setSelected(true) on it, one must first call setSelected(false) on the previously selected ListItem. Fig. 9 is a piece of sample code that violates this constraint.

Fig. 10 and 11 are constraint specifications written in LACOIO and Tracematch respectively. Both specifications define the same number of events (or symbols in Tracematch). But in Tracematch users are required to write a quite verbose regular expression. In LACOIO, we allow user to define a simple variable "$selected" to store the current selected ListItem. So with our approach we only need to check whether $selected is null when a second ListItem is about to be selected. However, with Tracematch, a pattern of events must be expressed in a temporal manner as regular expression, which is actually not necessary for verification. For example, the information of the temporal order of “select”, “get1” and “get2” is not necessary for identifying a violation. All that is
three automaton instances that are derived from the specified regular expression. But in LACOIO, monitor, <i1, i2, i3> are verified in one monitor instance, which avoids unnecessary verification and decreases the complexity.

VI. RELATED WORK

Tracematch [1] is a seamless extension of AspectJ and is a new history-based language. The logic that it relies is regular expression. It introduces free variables in the matching patterns. At runtime, the monitored objects and other values can be bound to the declarative variables so that the event can be matched not only by event kind but also by the value that associated with the free variables. MOP [5] is a formal framework for software development and analysis, in which the developer specifies desired properties using definable specification formalisms, along with code to execute when properties are violated or validated. It supports several logic formalisms, including CFG, FSM, regular expression, etc. MOP allows specifications with parameters. The free variables in Tracematch and parameters in MOP allow user to specify some COIOs. However, since the free variables or parameters are limited to a fixed number, when they are applied to verify some complicated COIOs that involve multiple objects of the same class like SafeDropStringLength, the specification will become verbose and hard to understand. LACOIO can lower the complexity of such specification and its monitor leads to a better performance for this type of COIOs.

Jaspan [2] proposed a lightweight specification system which can describe framework constraints that are temporal and involves multiple objects. Main differences between work in [2] and ours are: (1) the former is designed for static analysis. Some relations among objects are predicated so that they are not precise. The predicted relations are used for further analysis which makes the result unsound. LACOIO in this paper does not make any prediction. The verification is based on real runtime information of objects’ behavior and its result is sound (but not complete). (2) LACOIO is more flexible as it introduces user-defined variable, actions, etc, while the language in [2] only allows very simple pre/post-conditions, which makes it not applicable to specify constraints like SafeIterator and SafeHashMap[13].

Yahav [4] proposed an approach to verify safety properties using separation and heterogeneous abstractions. They presented a language for specifying separation strategies for decomposing a single verification problem into a set of sub-problems and then utilized heterogeneous abstraction during the verification of the transformed verification problem. However, this is not a general approach to describe many state-of-the-art API constraints. Moreover, this approach is designed for static program analysis, which is not appropriate for runtime verification of COIO.

VII. CONCLUSION

This paper proposed a novel specification language LACOIO to describe complex constraints on multiple interacting objects. We presented the language design and the implementation of the language compiler and the runtime verification framework. We have demonstrated that with LACOIO, we can describe complex COIOs in an easy and succinct manner and our language design has led to improvement of performance.

ACKNOWLEDGMENT

This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the Science Fund for Creative Research Groups of China under Grant No. 60821003; the National Natural Science Foundation of China under Grant No. 61033006.

REFERENCES