Fex: A Model Checking Framework for Event Sequences

Xin Li

Table of Contents

Chapter 1

Introduction

1.1 Statement of Research Problem

Program verification is the process of proving or disproving the correctness of an implementation with respect to a certain specification. It can either be specifically aimed at verifying the code itself, or an abstract model of the program. The proving process is usually done by providing a formal proof or, when disproving it, by providing further debugging information. Two approaches are typically used in program verification: theorem proving and model checking.

The theorem proving technique is based on deductive logic. A statement declaring that the program specification is satisfied by an implementation is formalized as a logic formula. Based on deductive rules from that logic, a theorem prover can be used to prove or disprove the formula. Theorem proving has the advantage of being able to handle programs with infinite states, but in most cases it is only partially automated and needs human guidance.

In comparison, model checking can be used with full automation because it verifies a system by exhaustively searching all possible states that a system could be entering. It is widely used for automatic verification of hardware or software systems. When doing model checking, the model checker either confirms that the given properties (usually specified as temporal logic formula) hold or reports that they are violated. In the latter case, it provides a valuable counterexample which can be used to find the error.

Since model checking is based on exhaustive exploration of all possible system states, if the number of program states is too large to be explored in a reasonable time, the model checker will react as dead without providing any useful feedback. This is called the *state explosion problem*. This problem is worse when conducting software model checking because software systems always manipulate very large data sets which may magnify the number of program states. In order to make model checking practically feasible, it is essential to reduce the size of programs and models to a manageable level.

Model checking is a feasible technique for verifying control-intensive properties because

these properties only needs limited data. In software systems, there are certain program properties which are control flow intensive. For example, when verifying a resource leakage problem, we only care whether the opened resource is eventually closed; how that resource is manipulated in computation is not of interest. However, a proper program abstraction process is still needed to make the model checking practical.

In this thesis, we focus on model checking those event sequence related program properties in which data are less important; that is, we do not care what the actual data value is; we only care if certain events are executed at the right place at the right time. Still, there are two main obstacles in applying this approach:

- How do we detect all possible system states including those states introduced by implicit control paths such as exception raised control paths?
- How do we handle the state explosion problem?

Exception handling is the main source for generating the implicit control flow. In most advanced programming languages like Java, exception handling provides a structured way for detecting and recovering from abnormal conditions. Some examples of abnormal conditions are: data corruption, precondition violation and environmental errors. If these abnormal conditions are not properly handled, they can cause an error or failure of the system. It has been reported that failures due to exceptions are estimated to account for two thirds of system crashes and fifty percent of system security vulnerabilities [62].

Generally speaking, it is difficult to protect a system from the effects of abnormal conditions because, by nature, all unusual occurrences cannot be anticipated when the system is designed. That is also the reason many security vulnerabilities are caused by exception related attacks. From this point of view, every program verification tool should consider the impact of exception handling on the system.

For the state explosion problem, most researchers agree that the best way to attack it is by program abstraction. Program abstraction constructs an abstract model based on the original program. On the one hand, the abstract model is built to be small enough to make model checking feasible. On the other hand, it is constructed to be large enough to capture all property-relevant behavior. To make model checking practical, efficient abstraction criteria should be provided for each checkable property.

This thesis focuses on how to solve these two main obstacles. We present a model checking based program verification framework on which several techniques have been deployed to solve these problems. The feasibility and effectiveness of this framework are demonstrated by three case studies.

1.2 Overview of the Work

We present a model checking verification framework for event sequence related program properties. The verification process can be divided into three steps:

- First, the investigated program undergoes static analysis and is instrumented so that all exceptions can be raised. Since we are concerned about event sequence related program properties that is indeed control flow determined, presenting implicit control flow such as exceptions are essential for the verification process. By using static analysis and program instrumentation, all exceptions can be presented in our framework with full automation. This gives us some confidence that our verification is complete.

- We then deploy an aggressive program slicer to perform a source to source program transformation. The program slicing process not only removes irrelevant program constructs but also conducts data abstraction, loop abstraction and predicate abstraction. Compared to the original program, the sliced program has a significantly smaller state space to explore.

  General speaking, we say a program P is event type T relevant with respect to a certain temporal logic property S, if and only if when with an abstract program P_a which preserves only control flow related type and event type T from program P, we have $P \vdash S \Leftrightarrow P_a \vdash S$. For certain event type T and program property S, program P can be safely abstracted to program P_a. Our framework defines program slicing rules that can transform the original program P into P_a where the program P_a is still executable but with fewer program states. Thus model checking program P_a is considered to be much easier.

- Finally, we adopt the sliced program as our program model and fed it into the back-end model checker. The model checking process can systematically exhausts all possible execution paths for the given simplified Java program. If there is any property violation, the model checker dumps out an execution path leading to the error.

We use executable specification to specify event sequence related program properties. Our executable specifications are written in Java and is used to regulate the designated behaviors of program events. Each event under investigation needs to have an executable specification counterpart. The new implementation, while executable, only specifies the event property constraints.

In this thesis, we present a model checking based verification framework, called Fex. Fex is a semi-automatic tools in the sense that during the whole process a manually prepared configuration file supervises the verification. This file provides critical information which is essential for the verification process. This information includes things such as the types of

events that are considered as relevant, the types of exceptions that can be ignored and the scope of the whole application that we should verify. In fact, this configuration file is used to build up the abstraction function to guide the instrumentation and program abstraction.

Three case studies have been done to demonstrate how our framework can be applied, and whether it can be helpful in practice:

- Exception reliability verification
- API conformance verification
- Java access rights verification

These properties are all event sequence related program properties. For exception reliability, the corresponding events are exception raising and handling. For API conformance, the corresponding events are these API methods we are interested in. For access rights verification, corresponding events are permission granting and checking operations. Therefore, these tasks can be handled by Fex.

False alarms are always a big concern for verification tools. Theoretically, Fex may generate false positives due to its over-approximation of the behaviors of the concrete program, e.g. the program abstraction techniques we adopted may introduce program behaviors which may not be presented in the original program. However, in our experiential use of Fex, no false positives are reported.

1.3 Summary of Contributions

This thesis is motivated by the desire to have an automatic tool for detecting event sequence related program errors. More specifically, it proposes a model checking based framework for Java programs to detect potential erroneous event sequences. A summary of the main result and contributions are as follows:

1. The design and implementation of Fex forms a large fraction of this work. We present a novel, property-guided, program abstraction technique which conduct both data abstraction, predicate abstraction, loop abstraction and program slicing to produce a simplified program for checking. We also integrate a program exception instrumentation procedure into our verification process to ensure no implicit control flow is missed.

2. We use executable specifications to describe event sequence related program properties. Previous widely used formalisms like finite state machines can be automatically translated into executable specification and executable specifications can be used for describing more complex program properties such as concurrent modifications and

resource leak problems. Executable specifications also provide as an extra program abstraction technique that lets us focus on what we are really interested in.

3. The feasibility and the potential usefulness of the approach are demonstrated by applying Fex to several verification tasks. Both the advantages and limitations of our methodology learned from the experiments are reported. These lessons are important for both the use and future development of the Fex tools.

The rest of this thesis is organized as follows:

- Chapter 2 provides the necessary background for understanding our work. We first introduce the general idea about exception handling and the exception handling mechanism adopted in Java programming language. The basic ideas of model checking technique are surveyed along with a brief introduction to the Java Pathfinder model checker. Finally, we describe some state of the art program abstraction techniques.

- Chapter 3 presents the design and implementation of Fex, which is used for exception reliability verification. The program exception instrumentation process and the program abstraction techniques adopted by our program slicer are introduced along with the final model checking process. Appendix A presents the technical descriptions for the Fex implementation and Appendix B gives a short description for the redundant exception handler detector which is based on basic Fex.

- Chapter 4 presents the extended Fex, which is use to verify event sequence related program properties. The Fex executable specification which is used on event sequence based program specification is introduced and an updated program abstraction technique is presented. The advantages and limitations of our methodology are also discussed.

- Chapter 5 and 6 present two case studies on the application of extended Fex . Chapter 5 is focused on API conformance verification, for example, to demonstrate that certain API is correctly used in a given application. Chapter 6 describes how Fex is used for Java access rights verification. That is, to verify if there is any possibility that an attacker can access data or services without proper permissions.

- Finally, Chapter 7 summarizes the main contributions of this thesis and outlines future work.

Chapter 2

Background

2.1 Exceptions

Exception handling is introduced in programming languages as a structured way for detecting and recovering from abnormal conditions such as: data corruption, precondition violation and environmental errors. If these abnormal conditions are not properly caught and handled, they can cause an error or failure of the system. Exceptions can help the programmer in simplifying the program structure and handling errors systematically. However, the use of exceptions can also be error prone, leading to new program errors and making the code hard to understand. For example, unrestricted use of exceptions often leads programs into an unanticipated state which is difficult to recover from. Also, the new control flow constructs introduced by exception handling may be easily misused. It is an important aspect of program correctness that a program is guaranteed to be reliable under exceptional conditions. A common solution to this problem is using reasoning techniques about programs to guarantee that all exceptions are handled properly.

2.1.1 Exception Handling Overview

Before exception handling features were added to programming languages, the common programming techniques used to handle abnormal conditions were return codes and status flags [13]. The return code technique requires each routine to return a value on its completion. Different values indicate if a normal or rare condition has occurred. The status flags technique uses a shared variable to indicate the occurrence of a rare condition. Setting a status flag indicates that a rare condition has occurred. Both techniques have noticeable drawbacks. First, the programmer is required to constantly test the return values or the status flags and there is no guarantee that all the exceptions will be handled eventually. Second, the exception handling code blocks are scattered throughout the code, which makes the program hard to read and maintain. Third, the return values and status flags provide only partial and local information about the erroneous situation, which makes recovery op-

eration a difficult task. In this section, we will introduce exception handling concepts which can alleviate all the above problems.

Terminology

We base our exception related terminology on the work by Goodenough [39] who introduced the exception handling concepts in use today.

Exceptional conditions are abnormal states in the execution of a program. When handled improperly, they may lead to system failure or crash. An exception is *raised* by the system when such an abnormal state is detected. The *Exception raise point* is the place where the exception is raised. *Exception handling* is a programming language construct designed to handle run-time errors which may occur during the execution of a computer program. An exception is *handled* when a certain block of code associated with that exception is executed. That block of code is called an *exception handler*. A code fragment that is labeled has a certain scope and is followed by corresponding exception handlers is called a *guarded block* or *protected region*. When there is no suitable handler for the raised exception, the exception must be handled by a default exception handler or passed to the caller. This mechanism is called *exception propagation*.

Example in Figure 2.1 demonstrates `Java`-style exception handling. `IOException` and `FileNotFoundException` are examples of exceptional conditions. The statement at line 5 is a possible exception raise point for exception `FileNotFoundException`. The method `readLine` at line 6 is a possible exception raise point for exception `IOException`. Line 9 to line 12 is an example of exception handler. This handler can handle `FileNotFoundException`. Code fragment from line 2 to line 8 which is surrounded by `try { }` is an example of a guarded block. Since there is no exception handler for `MyException` that is explicitly raised at line 7, this exception will be propagated to the caller.

Classifying Exceptions

Goodenough [39] classified exceptions into *domain failures* and *range failures*. Domain failure occurs when a precondition of an operation is violated. For instance, the operation is expecting a number but got a string instead. To deal with domain failure, the callee must be given enough information about the failure so that it can modify the inputs to satisfy the preconditions. If the callee is unable to fix the problem, it should be permitted to report the problem to the caller or simply terminate the whole program. Detailed information about the failure and the current execution stack should be preserved.

Range failure occurs when an operation either finds it is unable to satisfy its post-conditions or decide it may never be able to satisfy its post-conditions. For example, a read operation does not satisfy its post-condition when it finds an end-of-file mark instead of a

```
0  public void test() throws MyException {
1  ...
2   try
3   {
4     ...
5     BufferedReader br = new BufferedReader(new FileReader("MyFile.txt"));
6     String line = br.readLine();
7     if (cond) throw new MyException();
8   }
9   catch(FileNotFoundException e)
10  {
11    System.out.println("File MyFile.txt not found.");
12  }
13  catch(IOException e)
14  {
15    System.out.println("Unable to read from MyFile.txt");
16  }
17 }
```

Figure 2.1: A Java-style exception handling example

record to read, or a file opening operation can't satisfy its post-condition when after several attempts, it still can't get the correct file-name as the parameter. To deal with range failure, one needs the ability to abort the current operation or even terminate the whole program. Sometimes as a side-effect of terminating the operation, it is necessary to undo all effects of attempted steps. Under some circumstances, when the failure type and the corresponding solution are well known, the ability to try the operation again is also needed.

Leino and Schulte [57] consider Goodenough's categories of exceptions to be sometimes confusing. They replace the term domain failure with *client failure* and range failure with *provider failure*. Furthermore, they divide provider failure into two subclassifications, *observed program error* and *admissible failure*. Observed program error refers to the situation where intrinsic program error such as array out-of-bound or out-of-memory errors occur. In general, there is little that can be done to repair the situation when one of these errors is thrown. The common reaction would be termination of the program and reporting details about the exception and current execution stack. Under some circumstances, the ability to unroll the operation may be required.

On the other hand, *admissible failure* refers to an exception where one can recover from the failure state. For instance, an operation is intended to read bits from a network channel but the received bits contain too many parity errors. When one of these errors is raised, enough information about the error is usually available to make the recovery job feasible. This mechanism is used often when failure is expected–but only rarely–so that one can separate the normal handling from the exceptional ones.

2.1.2 Exception Handling Mechanism in Java

As an important control flow technique in language design, an exception handling mechanism needs to meet several general objectives:

- Provide built-in mechanisms to change the control flow when an exception is raised in order to avoid extensive testing of return values and status flags.
- Categorize all the exceptions into groups and hierarchies to make them easy to manage.
- Provide a mechanism to prevent any incomplete operations and thus keep the system in a consistent state.

In this section, we discuss the exception handling mechanism adopted in Java [4].

In programming languages, there might be several suitable exception handlers for a raised exception. Only one should be picked and executed. This mechanism is called *handler binding*. Java adopts semi-dynamic binding mechanism. The local handlers are bound statically to the guarded block. If no handler is found for the exception locally, the exception will be transferred automatically to the higher level caller along the control stack until a proper handler is found.

After the exception is raised and the corresponding handler is executed, the program should continue its normal execution. This is a semantic issue concerning how to determine the continuation of the control flow. Java adopts the termination model and uses `try/catch` keyword to define guarded region and exception handlers. For the termination model, the control flow transfers from the raise point to the corresponding handler according to the handler binding policies. When the execution of the exception handler is completed, the control flow continues as if the incomplete operation in the guarded block terminated without encountering any exception.

In Java, only subclasses derived from class `Throwable` can be considered as exception type, see Figure 2.2. The class `Throwable` has two predefined subclasses `Error` and `Exception`. Subclass `Exception` also has a predefined subclass `RunTimeException`. All subclasses derived from `Throwable` can be categorized into two groups: The first group contains these classes derived from either class `Error` or class `RunTimeException`. They are called *unchecked* exceptions. The second group contains these classes derived from either class `Exception` or directly from class `Throwable`. They are called *checked* exceptions. Using Leino's terminology, *unchecked* exceptions are observed program error. They include those exceptions that ordinary programs are not expected to recover from (for example, virtual machine error, null pointer error, etc.). *Checked* exceptions are actually admissible errors, they represent those abnormal situations with enough information to recover.

Software systems should be kept in a consistent state, no matter whether the code completes normally or is interrupted by an exception. In this sense, it is required to perform

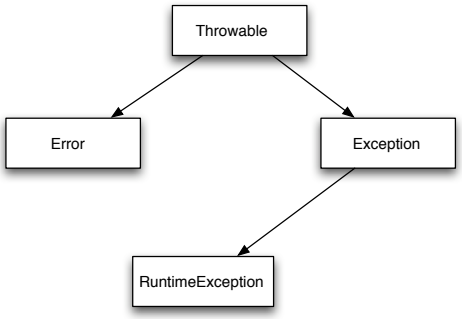

Figure 2.2: Exception Class Hierarchy in Java

some clean up actions to keep the program in a consistent state before the termination of the system. Clean up code should release allocated resources, undo some undesired operations or restore the program. Java provides programmers with the `finally` syntax feature which can be used to define clean up actions. The semantics of `finally` is: the finally block will always be executed at the end of the corresponding guarded block, no matter whether the exception is raised or not. In the case where the guarded block raises an exception that is not caught by its local handlers, the exception is propagated but the `finally` block is still executed prior to the propagation. For example, in the following code, no matter what happens in the `try` block and the corresponding exception handler, the resource `stream` will be eventually released due to the semantics of the `finally` block.

```
public void someMethod(File file) throws Exception
{
  FileInputStream stream = new FileInputStream(file);
  try {
      // process stream contents
  }
  catch (Exception e) {
      // do something
  }
  finally {
     if (stream != null) stream.close();
  }
}
```

2.2 Model Checking

Model checking [21, 50, 64] is the process of checking whether a given model satisfies a specified property by exhaustively searching all possible states that a model could enter. The system under investigation is often modeled as a state transition system $\mathcal{M}$, where nodes represent the reachable states of the system and edges represent state transitions. The properties that designers wish to impose on the system are presented as logical formula ψ which usually fall into two categories [56]:

> **Safety:** properties that state "something bad never happens". A system satisfies such a property if it does not engage in the proscribed activity. For instance, a typical safety property for a coffee-machine is that the machine will never provide tea if coffee is requested by the user.
>
> **Liveness:** properties that state "something good will eventually happen". To satisfy such a property, the system must engage in some desired activity. An example liveness property for the coffee machine is that it will provide coffee eventually after a sufficient payment by the user.

Although informal, this classification has proved to be rather useful [53]. The two classes of properties are almost disjoint, and most properties can be described as a combination of safety and liveness properties. Logics have been defined to precisely describe these type of properties. The most widely studied are *temporal logics* [32] which support the description of system behaviors over time. Variants of temporal logic differ in the connectives they provided and the semantics of these connectives. The main temporal logics used currently are CTL [19], LTL [60, 61] and CTL*[18].

In order to model check a system, the system should first be defined by using the definition language of a model checker, thus providing the model $\mathcal{M}$. Then the program properties should be coded by using the specification language of the model checker, thus providing the logic formula ψ. The model checker then performs a check on all behaviors of the model. The result of the model checking process either confirms that the properties hold or reports that they are violated. In the latter case, it can, where one exists, provide a counterexample which provides valuable feedback illuminating the error. Compared to theorem proving techniques, model checking has the advantage of being automatic and cost effective, albeit it is limited to finite systems with relatively few states.

There are two types of model checking:

- **Explicit state model checking:** Originally, the algorithm for solving the model checking problem represents the transition system explicitly as a labeled, directed graph and is therefore called *explicit state model checking*. The algorithm performs

a search strategy (usually depth first search) through the states of the graph. The search is performed in the forward direction state by state (some reduction techniques can be deployed to skip some states), from a state to its successors, unless the search strategy requires backtracking. Thus, it is straightforward to assess the number of states encountered during the verification. Since a transition system could contain millions of states, abstraction techniques are essential for the explicit state model checking.

- **Symbolic model checking**: For explicit state model checking, although the model checking algorithm is linear in the size of the model, the model size itself can be exponential in the number of the variables, which is called the state explosion problem. Efficient data structures such as the Ordered Binary Decision Diagram (OBDD) are used to attack this problem. In symbolic model checking[63], the transition system is encoded by using ordered binary decision diagram (OBDD) [12].

  A OBDD is a directed, acyclic graph which is used to represent a Boolean function. Each state of the transition system is encoded using OBDD variables. A set of states is encoded by a boolean function and transition relations are encoded as sets of pair states. OBDDs have become very popular for model checking because they offer the following features:

  - OBDDs are often substantially more compact than traditional representations of transition systems.
  - Every boolean function has a unique, canonical OBDD representation so that state equality check can be performed in constant time.
  - Boolean operations such as negation, conjunction, implication etc. can be implemented with complexity proportional to the product of the inputs.

In general, the model checker requires a closed system to analyze which means the system and the according environments that the system may manipulate must be provided before conducting the model checking. However, the environment is often unavailable and therefore needs to be created. Currently, the model checker user often needs to construct the environment by hand. Since manually presenting all relevant environment is an error-prone job and sometimes needs domain knowledge (which is impossible to obtain if the domain expert are not involved), tools and methods are needed to assist in the construction of program environments.

2.2.1 Software Model Checking

Since model checking essentially proves or disproves a property of the system by exhaustive enumeration, it was first applied to small systems, protocols and hardware related verification. The method has been used successfully in practice to verify complex sequential circuit design and communication protocols [31]. Inspired by these applications and increased computing power, there is a new trend to use model checking techniques to verify real software systems via so-called software model checking [43].

Conventionally, model checking software systems is primarily concerned with design validation. That is, one first builds a formal model based on the software design, and then this model, combined with a set of temporal formulas which reflect the desired software properties, can be fed into the model checker. The model checker can automatically validate the design or produce a counterexample which is a trace of erroneous behavior. Once the design is validated, the actual implementation can be written based on this verified design. In this process, software errors are caught earlier in the development life cycle.

However, verifying software design only is not enough. There is still a big gap between the software design and the final product. The final implementation may still have subtle errors despite the existence of careful design. In fact, errors such as deadlock are often introduced at the coding level where designs typically do not cover. In recent years, there has been increased interest in model checking the software program itself. Several software model checkers have been developed such as Spin [46], Bandera [23], Bogor [72], JPF [58], Verisoft [38], Modex/Feaver [48] , Zing [3] and Slam [5]. Some of them have been successfully applied into industry level program verification and subtle program errors have been identified [69, 10, 14].

Model checking of software written in a general purpose programming language can be done in two ways:

- Extract the model from the program, then verify the model with a model checker.
- model checking the program directly.

Each choice can have a number of implications. By translating the program to a modelling language acceptable to existing model checkers, we avoid the need to re-implement the existing model checking techniques. However, the limited expressive power of the existing modeling languages often makes this process more difficult than it sounds. Many language constructs of general purpose programming languages cannot find a counterpart in the modelling languages and hence need special treatment. For instance, for these features that are difficult to translate, Modex/FeaVer requires the user to define the translation/abstraction rules. Otherwise the default action will simply put the code in print statements which are

not part of the model. By doing that, the safeness and completeness of the model checking process can no longer be guaranteed.

Adopting the second approach obviously requires us to build everything from scratch. However, building the new custom-made model checker can have the advantage of high coverage of the destination program language features. It also gives us the opportunity to invent and experiment with new model checking techniques that may be suitable for software model checking. Java Pathfinder (JPF), for example, uses its own Java virtual machine and therefore can handle all the language features of the Java program. Is is also an ideal experiment platform for various program abstraction techniques. Our verification framework Fex uses JPF as the back-end model checker.

2.2.2 Java Pathfinder Model Checker

Java Pathfinder (JPF)[43] is a software model checker which handles Java byte code directly. This model checker is different from others in the sense that it consists of its own Java virtual machine (called JVM^{JPF}) that executes the byte code and a search component that guides the execution. By modeling the snapshot of the JVM^{JPF} into a concise state representation, one can use explicit model checking algorithm to systematically explore all the potential execution paths of a program to find out potential errors. So far JPF can search for deadlocks, unhandled exceptions and assertion violations, but the user can use JPF's extension mechanism to write their own property classes.

The state explosion problem, a common problem associated with explicit model checking, is tackled by JPF in several ways.

- JPF adopts the state collapse method [47] for storing complex virtual machine states and uses an efficient hash function for fast state comparison.
- The virtual machine of the JPF is carefully designed so that symmetry reduction [51] and partial order reduction [17] can be used to reduce the system states.
- Several abstraction mechanisms such as Model Java Interface (see below) is supported by JPF.

JPF tries to overcome the systematic scalability problem of software model checking by providing application or property specific abstractions. As a consequence, it provides two major extension mechanisms, namely Search/Virtual Machine Listeners and Model Java interface (MJI).

- JPF provides two `Search/Virtual machine` Listeners allow the user to communicate with the JPF internal state model and get notified when special events happens. This is achieved by using the Observer pattern [37] that lets the concrete observers (listeners) to subscribe to certain events inside JPF. These events can be a specific byte code

instruction execution or the search forward/backtrack steps. Search/Virtual Machine Listeners provides a convenient way to add new property checks or change search policies. For example, if we are interested in verifying that there is no way that an `FileNotFoundException` could be raised, we can rewrite the event `exceptionThrown` from `VMListener` so that every-time an `FileNotFoundException` raised, the search stops and an error report is submitted. The new functionality is not limited to add verification properties. It can be anything that uses JPF's systematic program execution approach. For example, it can be used to produce code coverage metrics.

- JPF also provides a mechanism called Model Java Interface (MJI) to separate and communicate between the JPF controlled virtual machine (which is state-tracked) and the host virtual machine (which is not state tracked). MJI can be used for the following usages:

  1. MJI is mainly used to intercept native methods. Since JPF cannot reason about native calls and therefore it must have models of these native methods instead. These models are created using the MJI and are used to replace the actual native method. The model developer is responsible for abstracting the important characteristics of the native call and retaining it through the model. JPF can then reason about the behavior of the native call by using these models and MJI mechanism will make sure that the JPF is checking the model instead of the original native method.
  2. MJI can be used to do the state space reduction. For some methods such as `System.out.println()`. We know that its execution has no side effect on the property we want to verify. we can use MJI to delegate the corresponding bytecode execution into the non-state-tracked host JVM which can reduce the state spaces need to be explored.

JPF needs a closed system (a system and all the environment it will execute in) to analyse. The current version of JPF can handle all the Java language features including commonly used libraries. It also treats non-deterministic choice expressed in annotations of the program being analyzed. These annotations are added to the programs through method calls to a special class `verify`. For example `Verify.getBoolean()` returns a non-deterministic boolean value. Once an error is detected, JPF reports the entire execution path that leads to the defect.

So far JPF can check program properties like deadlocks, unhandled exceptions and assertion violations. It has been effectively used to find errors in a number of complex systems including the real-time operating system DEOS from Honeywell [68] and a prototype Mars Rover developed at NASA Ames [11].

2.3 Program Abstraction

Since software system always manipulate very large data sets, the state explosion problem is more immediate while doing software model checking. In order to make model checking practically feasible, it is essential to reduce the size of program models to a manageable level. It is also essential that the behaviors of the reduces program/models, with respect to the properties being checked, is exactly the same as the original program/models. Advanced model extracting techniques must be applied to satisfy the above two requirements. In general, the goal of these techniques is to reduce the state space of the system that the model checker needs to analyze, while still preserving the interesting program behaviors.

Most researchers agree that the best way to attack the state explosion problem for software model checking is using *program abstraction*. Intuitively, program abstraction is a general approach which allows one to deduce properties of a concrete program by examining a more abstract and smaller program model. This abstract model should be small enough to make automatic checking tractable, yet it should be large enough to capture all information relevant to the property being checked.

Abstractions can be performed either on program model level (labeled transition system) or on program source code level. Abstraction techniques on transition system are partial order reduction [17], symmetry reduction [51], parametrization, cone of influence reduction [21, **?**] etc. Because the space space for a even small program might be extremely large, it may not be possible to build a transition system for a reasonable sized program. On the other hand, abstraction on the program source code directly by program transformation scales well with the program size and is therefore widely used in software model checking process. In this section, we present several program source code abstraction techniques such as abstraction interpretation, data abstraction, predicate abstraction and program slicing. These techniques have been successfully deployed in several software model checkers. In order to make software model checking more feasible, new abstraction techniques must be investigated.

2.3.1 Abstract Interpretation

Abstract interpretation [25] is the technique for formally building a conservative approximation of the semantics of programming languages. An abstract interpretation is defined as a approximated program semantics obtained from the concrete program languages by replacing the concrete domain and its concrete semantic operations with an abstract domain and corresponding abstract semantic operations. Abstract interpretation over-approximate the behavior of the concrete program so that every behaviors of the concrete program is covered by a corresponding abstract execution.

We present a common abstract interpretation method to provide a flavor of the technique.

Sign abstraction consists of replacing integers by their sign and ignore their real values. The concrete domain `integer` is now replaced by the abstract domain `{pos, zero, neg}` All element operations on the concrete integer domain is redefined. For some element operations, such as multiplication, such an abstraction does not lose any precision: to get the sign of a product, it is sufficient to know the sign of the operands (see Figure 2.3 part (b)). For other operations, the abstraction may lose precision. This means that abstract interpretation may introduce extra behaviors namely behavior which does not occur in the concrete program. For example, the concrete result for operation $x + y$ when $x = -2$ and $y = 3$ is 1. However, according to Figure 2.3 the result from the abstract domain is undecided (see Figure 2.3 part(a)).

+	neg	zero	pos
neg	neg	neg	pos/zero/neg
zero	neg	zero	pos
pos	pos/zero/neg	pos	pos

(a) Addition

*	neg	zero	pos
neg	pos	zero	neg
zero	zero	zero	zero
pos	neg	zero	pos

(b) Multiplication

Figure 2.3: Abstract interpretation example

Another common abstract interpretation is *interval abstraction* that approximates a set of integers by its maximal and minimal values. Thus, if a counter variable appears in a property, the counter can be replaced by the lower and upper bound limits of the counter. Detailed introduction and treatment about abstract interpretation can found in [27, 24].

Since abstract interpretation is not specific to any given property or for any given program, they have the power of generality. However, the abstract version of the language semantics often need to be constructed manually, which is considered as tedious work.

2.3.2 Data Abstraction

Data abstraction operates directly on data values and data operations. By abstracting away some of the data information, data abstraction can create a smaller model. It is often performed on the program text manually.

We introduce several commonly used data abstraction techniques. The first one is the arithmetic operation based data abstraction. For verifying program involving arithmetic operations, congruence modulo a specific integer could be very useful. Thus, for any operation on integer i, i is now replaced by $i \ mod \ m$. Therefore, the original large concrete integer domain is abstracted down to m elements. Also, when comparing the orders of magnitude of some quantities, using the logarithmic representation to replace the actual value has proved to be useful.

Symbolic abstraction is used in situations where the enumeration of the data values is tedious. For instance, an application might store a set of items as a vector, but the

specification being verified may only depend on whether a particular item is in the vector or not. In this case, we can abstract the large number of vector states onto a small set `{ItemInVector, ItemNotInVector}`.

In all the above abstractions, the infinite behavior of the system, resulting from the presence of variables with large domains, is abstracted. The data abstraction is performed by mapping the data type of the actual system to an abstract data type. For each variable to be abstracted, the abstract domain and the operations are defined. An abstract model is then obtained by replacing each concrete variable and operation by an abstract one.

Data abstractions are very similar to the abstract interpretation technique. The difference between data abstraction and abstract interpretation is: Data abstraction does not always hold over all of the concrete system's execution semantics but abstract interpretation does. Every abstract interpretation framework needs to establish a methodology based on rigorous semantics for constructing abstraction that is guaranteed to over-approximate the behavior of the original program.

2.3.3 Predicate Abstraction

The *Predicate abstraction* technique was first introduced in [74]. The basic idea of predicate abstraction is to replace a concrete variable by a boolean variable that evaluates to a given boolean formula (a predicate) over the original variable. This concept can be easily extended to handle multiple predicates and predicates over multiple variables.

We use the programs in Figure 2.4 as an example to illustrate the idea. In this figure, we have a program with three integer variables x, y and z, which can grow ambitiously large and therefore make the program state-spaces practically unmanageable.[1] However, close inspection may reveal that the only concern about these three variables is their relationship to each other. We can then define two predicates $b1 : x < y$ and $b2 : y < z$ to represent the relationships. These two predicates can then be used to construct an abstraction of the system's behavior as follows: wherever the condition $x < y$ appears, we replace it with the predicate $b1$, and whenever there is an operation involving x or y which may change the value of $b1$, we replace it with an operation changing the value of $b1$ accordingly. We do the same to condition $y < z$ with $b2$.

Early predicate abstraction applications [74, 7, 22] require the user herself to identify the predicate sets. This user-driven predicate discovery does not scale very well.

The SLAM project [5], quite uniquely, uses automated boolean abstraction as the basis of its model checking process. The model checking process can be divided into three steps:

- First, a C program is translated into a boolean program with respect to a set of predicates over the variables of the original program. The boolean program updates

[1] The symbol * in the program represents the non-deterministic choices according to the type definition.

```
01  void main()
02  {
03   int x = *;
04   int y = *;
05   int z = *;
06   if (x<y) {
07      if (y<z) {
08         error();
09      }
10    }
11   }
```

(a) Original program

```
01  void main()
02  {
03   bool b1, b2;
04   b1 = *;
05   b2 = *;
06   if (b1) {
07      if (b2) {
08         error();
09      }
10    }
11   }
```

(b) Abstracted program

Figure 2.4: Predicate abstraction example

its variables by finding how executing the state would affect the truth of the predicates.

- Second, a model checker exhaustively explores the state space of the boolean program. If the checker hits a error, we have an abstract counterexample.
- Third, we check if the abstract counterexample corresponds to a real program error. If not, the predicate sets achieved in the first step are refined. We then repeat the process.

This kind of program abstraction process is called *Counterexample-guided abstraction refinement*. The validity of SLAM toolkit has been demonstrated by model checking two properties on a number of windows NT device drivers. In all these cases, the properties were validated within just a few iterations. Another model checker which is based on predicate abstraction and counterexample-guided abstraction refinement is the BLAST [45] project.

2.3.4 Program Slicing

Program slicing [41] is a technique that reduce the size of the programs by eliminating statements that are irrelevant to the designated program properties. Program properties that must be preserved are defined by *slicing criteria*. Typically a slicing criterion consists a set of program points of interest. A slicing process is then defined by describing the slicing criterion and the transformation rules to be performed upon the program for constructing the slice. Program slicing has been widely used in testing, debugging, program maintenance, program comprehension, complexity analysis, reverse engineering and model checking.

An example of program slicing is illustrated in Figure 2.5 [2]. The original program in part (a) calculates the sum and the product of a natural number n. The slicing criterion is defined as $(10, product)$, which means we are only interested in variable *product* at line 10 and all other program statements which are irrelevant to this criterion can be removed. Part

[2]This example is borrowed from [77].

(b) shows the sliced program w.r.t. this slicing criterion. As we can see from this figure, all statements involving only the variable *sum* have been removed.

```
01  read(n);
02  i := 1;
03  sum := 0;
04  product := 1;
05  while i <= n do
    begin
06    sum := sum + 1;
07    product := product * i;
08    i := i + 1;
    end;
09  write(sum);
10  write(product);
```

(a) Original program

```
01  read(n);
02  i := 1;
03
04  product := 1;
05  while i <= n do
    begin
06
07    product := product * i;
08    i := i + 1;
    end;
09
10  write(product);
```

(b) Sliced program

Figure 2.5: Program slicing example

In [41], program slicing is performed by conducting the data flow and control flow analysis which compute the consecutive sets of indirectly relevant statements. Because only the static information is used in this process, it is called *static program slicing*. An alternative way for program slicing is proposed by [67], which is depend on the Program Dependence Graphs (PDG) [49]. A PDG is a directed graph with vertexes represents statements and control predicates, and edges represents to data and control dependencies. Using PDG, one can build the program dependent graph by using slicing criteria as the initial vertexes. The sliced program corresponds to all the PDG vertexes from which the initial vertexes can be reached.

Since program slicing can reduce the size of the programs and the corresponding program state spaces that need to be explored, it allows the model checker to handle larger programs. This technique has been used in both hardware model checking [20] and software model checking areas [65, 42].

When slicing for model checking, the slicing criterion are often related to the properties under analysis. For a given property P, the slicing criterion is the set of program points that affect the values of the variables presents in P. Therefore, every statement which might affect the slicing criterion should be preserved; otherwise the result slicer is not functionally equivalent to the original program w.r.t. the properties. The program slicing process in model checking can be seen to be a pre-processing phase which preserve every statement that might affecting the slicing criterion. More specifically, the slicer guarantees that:

- The sliced program should be functionally equivalent to the original program w.r.t. to the slicing criterion.
- The sliced program should still be executable.

The effectiveness of the slicing for reducing program size varies depending on the structure of the program. In some cases, slicing could effectively remove the irrelevant states and dramatically reduces the state space. In other cases, where large sections of the program are relevant to the specification, the effectiveness of slicing is negligible.

Chapter 3

Basis: a Framework for Verifying Exception Reliability

In this chapter, we first introduce the idea of exception reliability and the corresponding analysis tools. We then describe how our framework can be used for exception reliability verification.

3.1 Exception Reliability

Exception handling mechanisms in programming languages can help in simplifying program structure and in the systematically handling of errors. However, the use of exceptions can also be error prone, leading to new program errors and making the code hard to understand. For example, unrestricted use of exceptions often leads programs into an undetermined state which is difficult to recover from. Also the new control flow introduced by exceptions may lead to program errors such as resource leakage. It has been reported that failures due to exceptions are estimated to account for two thirds of system crashes and fifty percent of system security vulnerabilities [62]. A common solution to these problems is to use tools that guarantee that all the exceptions are handled properly.

We call a program "exception handling reliable" when the following conditions are satisfied:

- All possible exceptions are handled.

- There is no unreachable exception handler in the program.

In real applications, one might divide all the possible exceptions into levels of importance, and then decide on which levels these exceptions should be handled and on which they might not. For example, the Java language divides all exceptions into two categories. One is called *checked exceptions* and the other is called *unchecked exceptions* where checked exceptions are considered more important and must be handled by the programmer. However, for a robust program, all possible exceptions should be properly handled.

Unreachable exception handlers are another common mistake in programming. For example, one might write an exception handler for an exception that will never be thrown from the try block; or, if the exception representation is adopting a class hierarchy mechanism, one might write a handler for the subclass right after the handler for the super-class. In that case, the subclass exception handler will never be executed and therefore is unreachable.

3.2 Existing Tools

Most compilers can do some shallow checks like type checking to guarantee that every exception is carrying the correct parameters. Some language compilers, such as the Java compiler do more advanced checks for checked exceptions. The Java compiler can guarantee that all checked exceptions will be handled locally or propagated to their caller. However, compilers can not find all the possible errors mentioned above. Various static analysis tools and techniques have been proposed to address problems related to exception handling. Using static analysis, we can estimate the dynamic behavior of the system and gather the information about uncaught exceptions and the exception propagation flow. This information can be used to facilitate the understanding of exceptional program behavior and make better use of the exception handling mechanism. These static analysis tools may generate false positive alerts since they can only simulate the run-time behavior of the exceptional control flow. A detailed review of static exception analysis techniques and corresponding tools is available in [15].

Historically, exception analysis was first introduced for ML based on abstract interpretation [82], which is shown to be very slow. So the analysis was redesigned [83] based on a set-constraint framework to improve the speed, and the implementation is integrated in the SML/NJ compiler to give programmers information on potential uncaught exceptions. In [16], an efficient inter-procedural exception analysis was proposed by applying the idea in [83] to Java so that one can estimate uncaught exceptions independently of the programmer's specified exceptions. They implemented the analysis on top of Barat, a front end of a Java compiler. It has shown that the analysis is able to detect uncaught exceptions and unnecessary `catch` and `throws` clauses effectively. Robillard and Murphy [73] also have developed a similar tool called Jex for analyzing exceptions in Java.

Among all these static analysis tools, Jex is a typical representative. Based on a general model of the exception-handling structures and algorithms [75], the Jex tool extracts the information about the structure of exceptions in Java programs, and provides a view of the actual exception types that might be raised at different program points. Using this information users can detect uncaught exceptions, find out redundant exception handlers and get a better understanding of the system structure. All these benefits can help users to build robust applications.

3.3 Introduction to basic Fex

3.3.1 An Uncaught Exception example

Figure 3.1 shows an example of a program with an uncaught exception. The program first creates a `FileInputStream` object which is connected to the file named `args[0]`. Then, it reads the contents from this file byte by byte (by executing `fis.read()`) and prints the contents on the standard output. When the end of the file is reached, the program closes the file and exits. The program is handling exceptions, but it skips the possible `FileNotFoundException` which could be raised by line 04. In a suitable environment, `FileNotFoundException` will be triggered and since there is no exception handler for this exception, the whole program is led to an abnormal exit; therefore, this program is not exception reliable.

```
01 import java.io.*;
02 public class Example1 {
03    public static void main(String[] args) throws IOException {
04        FileInputStream fis = new FileInputStream(args[0]);
05        try{
06            int i = fis.read();
07            while(i != -1) {
08                System.out.print((char)i);
09                i = fis.read();
10            }
11            fis.close();
12        }
13        catch (IOException ex) {
14            System.err.println(ex);
15        }
16    }
17 }
```

Figure 3.1: Example of Uncaught Exception

This uncaught exception flaw is quite trivial in the sense that it can be detected by an experienced programmer working with a standard Java run-time environment. First of all, a standard Java compiler will consider this program as "correct" because at line 3, the program explicitly declared that an uncaught `IOException` might be propagated. However, if an experienced programmer senses that there might be a tiny possibility that the required file is not available, he can then deliberately mimic this situation by setting the file to be unreachable. The JRE (Java runtime environment) will reply to this setting by reporting an error message like the following:

```
Exception in thread "main" java.io.FileNotFoundException: ... (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:106)
        at java.io.FileInputStream.<init>(FileInputStream.java:66)
        at Example1.main(Example1.java:4)
```

We would like to build a model checking based verification framework to verify that a given program is exception reliable. This verification framework should provide following

functionaries:

- The framework should be fully automatically. Thus, the exception reliability flaw should be detected without any extra man-made error conditions.
- The framework should handle not only uncaught checked exceptions but also uncaught unchecked exceptions. Although unchecked exceptions are often considered as trivial and therefore discarded by the standard Java compiler, we think, under some circumstances, some of them are important and need to be considered in the exception reliability verification.

3.3.2 Fex Structure

Model checking is based on exhaustively exploring all possible states of the model. In general, a model is given as a labeled transition system such as Kripke structure where nodes represent the reachable states of the system and edges represent state transitions. The model checking process can then be rephrased as traversing the whole system to find whether an error situation is reachable from the initial nodes. However, in the real world, the program is written in language such as Java which is based on variables, references, functions and threads. There is a big gap between the real program and the model checking model. Usually it is not easy to translate a real program into a finite state model which can be traversed in a manageable time.

The main obstacle in applying model checking techniques to software verification is the state explosion problem. All programs need to be abstracted to an finite-state model before feeding into the model checker. On the one hand, this model should be big enough to carry all interesting program properties. On the other hand, this model should be relatively small so that the whole model checking process is feasible.

We have developed a model checking framework, called Fex, which can be used for exception reliability verification. In order to alleviate the state explosion problem while still preserving all interesting program properties, Fex adopts several novel techniques.

The whole framework can be divided into three parts:

1. **A static analyzer for exception instrumentation.** The first part of Fex is a static analyzer. This analyzer is used to collect all exception related information. As we are conducting a exception reliability check that is indeed a control flow related program property, acquiring a complete program control flow is crucial. Our static analyzer can gather all exception related information and produce a new program with all possible exceptions instrumented at the proper places. With the help from this instrumentation, the back-end model checker can now be aware of these potential exceptions and check the corresponding implicit program control flows raised by these

exceptions. Thus we can now have a complete control flow of the program under investigation.

2. **A program slicer for program abstraction.** The second part of Fex is a program slicer for program abstraction. In general, the program states for a regular size program is intractable without any program abstraction. Our slicer performs a source to source transformation with three abstraction operations:

   - Our aggressive program slicing replaces a general loop construct with a fixed iteration loop. When reasoning about control flow related program properties, these control flow related program properties are typically dominated by the structure of the program control flow. As the program control structure is usually determined by a small set of control flow variables including guards for the conditional statements, exceptions and the loop indexes. Model checking such control flow related properties needs to traverse all these possible execution paths. For conditional statement, we need to traverse both branches. For exceptions, we need to explore all potential exceptions. For loops, we need to check every single iterations. However, because the program properties we are interested in are insensitive to the loop index, instead of executing all loop iterations, we can only execute a fixed iteration of the loop to see if there is any property violation. This replacement can minimize the contribution of the loop execution to the system states. Because exception reliability property is loop index insensitive, this transformation can preserve exception reliability property.
   - The program constructs which manipulates only "don't care" variables are removed. In the original program, there are only a small amount of variables such as control flow variables and program property related variables are considered as relevant to the property being checked. Other variables are treated as "don't care" variables. Based on this observation, we divide all program variables into several type groups (control flow related type group, verification property related type group, irrelevant type group) and all these "don't care" variables belong to the irrelevant type group. By conducting type analysis. our program slicer can slice away these program constructs which only manipulate "don't care" variables. Removing these parts can greatly reduce the program states which need to be explored.
   - Although the program constructs which manipulates only "don't care" variables are removed, there are still some "don't care" variables that need to be preserved due to the structure dependence. For example, for a method with a parameter which is actually a "don't care" variable, this parameter needs to be preserved.

These variables are only preserved to make the sliced program syntactically correct *i.e.* to make the sliced program executable. The real values of these variables are irrelevant. Therefore, we can stuff these variables with a predefined ("don't care") value to eliminate the data diversity. Because the data diversity is one of the main contributor to the state explosion problem, this diversity reduction action can eliminates many program states. Also, we can now skip the test case preparation phase (traditionally, these variables need to be stuffed with a meaningful value to conduct the verification).

Compared to the original program, the sliced program has a significantly smaller program state space to explore.

3. **A model checker for verification.** The third part of Fex is the back-end model checker JPF. We adopt the sliced program as our program model and feed it into JPF . The model checking process is conducted upon this simplified program on-the-fly in the sense that we only check reachable program states. Compared to model checking the whole transition system, the reachable state are often in a small fraction, which means we abstract away entirely exponentially large number of unreachable program states. JPF can systematically exhausts all possible execution paths for a given Java program. In our framework, it checks if there is any uncaught exception in the sliced program. If so, JPF dumps out an execution path leading to the property violation. This execution path can help the programmer to fix the problem. We selected JPF as our back-end model checker for a number of reasons:

   - Compared to other model checkers like Bandera [23] or Bogor [72], JPF can handle more Java features.
   - As an explicit state model checker, JPF adopts several efficient state reduction techniques like heap symmetry reduction and partial order reduction.
   - JPF provides non-deterministic constructs for modelling environment driven choices.
   - JPF is an open source project.

Since the model checker typically finds only one problem at a time, each iteration corrects a single problem and the process is repeated.

Besides these three main parts, Fex also need a configuration file to guide the verification process. This configuration file is used to provide extra information to support the verification. For example, for some applications, instrumenting all exceptions is tedious and impractical. We need to provide the static analyzer the extra information about which exceptions can be safety ignored during the exception instrumentation process. Also, for some more complicated verification tasks such as API conformance verification (see the

next chapter), we also need to provide the information about which API need to be verified. All of these data can be obtained from our configuration file. This file indeed specifies an abstraction function which is used to guide the program instrumentation and slicing.

3.3.3 Verification Process

The verification process in Fex is designed to be iterative, with each cycle divided into three steps (Figure 3.2). To illustrate the whole process, we use the program from Figure 3.1 as the sample Java program.

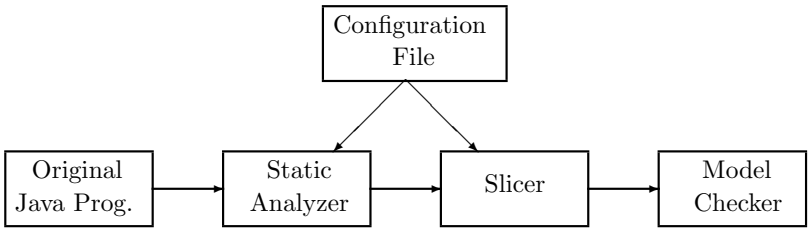

Figure 3.2: Verification Process

The first step is to use the static analyzer to collect the exception information and instrument them into the proper place in the original program so that all possible exceptions can be raised. The result for the exception instrumentation for program in Figure 3.1 is shown in Figure 3.3. Notice how lines 5–7, 9–11, 14–15, 17–18 in Figure 3.3 are changed to represent possible exceptions raised by lines 4, 6, 9, 11 from Figure 3.1. For detail about how this instrumentation is done, see Section 3.4.

The second step of our process applies the program slicer to do the program abstraction. For the exception reliability problem, we only preserve control flow related information such as threads, flow structures and exceptions. All other program constructs and variables are either discarded or stuffed with a predefined value. Every loop construct is replaced by a fixed iteration. Figure 3.5 presents the sliced version of the program in Figure 3.3. As we can see, the simplified program contains only control flow related information. For detailed information about how program slicer works, see Section 3.5.

The third step of our analysis feeds the sliced program into the software model checker, Java Pathfinder (JPF) [78], to search for any uncaught exceptions. Upon detecting a possible exception reliability violation, JPF dumps out an execution path leading to the uncaught exception which can help the programmer to fix the problem. This verification process is iterated until no further violation is reported. For the example from Figure 3.1, the model checker eventually reports that the possible `FileNotFoundException` raised by `new FileOutputStream()` operation (see Figure 3.6) inside the `main` method is not handled.

3.4 Instrumenting Exceptions

Exceptions are often caused by unanticipated or rarely encountered situations. They complicate the program control flow by creating implicit control flow paths within or across methods. As we are interested in verifying exception reliability, which is essentially a control flow related program property, how to acquire a complete control flow from the program is crucial. To be precise, all exceptions-triggered program control flows must be included in our verification scope.

Our verification is based on model checking. Although the back-end model checker has the power to check all explicitly stated control flow paths, how to exhaust all these implicit control flows such as exception introduced control flows is still a challenge.

Directly raising the corresponding exception is a convenient way for checking program behaviors under exceptional conditions. By providing proper data which can trigger exceptions, the corresponding program behavior under these exceptions can be model checked. However, this method has certain disadvantages:

- In order to trigger exceptions, the test harness needs to be carefully prepared. This process is tedious and error-prone.
- Performing the checking based on a small set in a test harness is indeed program testing, which is not as convincing as a formal analysis method.
- Data driven test harness can not trigger some environmental exception situations such as power failure.

Static code analysis is an alternative way to provide exception information. By approximate the program behavior at the run-time, a static analyzer can gather the exception related information such as what kind of exception that might be raised at different program points. This information can be used to build the overview of the entire exception structure of the program, which can help the developer to reason about the exceptional behaviors of the program.

We propose to combine static analysis together with model checking techniques to perform a complete program control flow check. A static analyzer is first deployed to record all potentially raised exceptions and the corresponding raising place. Then, we instrument all these exceptions at the exact place where this exception might be raised. By doing the instrumentation, the back-end model checker can have the power to exhaust all exception introduced implicit control flows.

The instrumentation of a Java file is accomplished by traversing the abstract syntax tree (AST) of the file. For each statement, the analyzer instruments possible exceptions according to the Java language specification [4]. For example, a statement a/b which attempts

a division operation might raise `ArithmeticException` and therefore is instrumented as follows:

```
a/b;
if (Verify.getBoolean()) throw new ArithmeticException();
```

Thus, not matter what the real value of a and b are, we always considered the possibility of divide by zero error and therefore raises `ArithmeticException`. Please note the instrumentation call `Verify.getBoolean()` returns a Boolean value non-deterministically. This ensures that in the future, the model checking explores all exception-triggered execution paths in the original program.

For each method call, we have to determine if there is any possible exceptions propagated from the method. The analyzer first determines if the method source code is available. If available, the analyzer instruments the method directly; the model checker can take care of the exception handling and propagating at run time. If not, the analyzer instruments the exception interface extracted from the byte code of that method to guarantee that the exception has a chance to be raised. For example, in Figure 3.1 line 9, although we do not have the source code for `FileInputStream` available (even if we have the source code, we would not plan to include it into our verification scope), from the corresponding byte code we know that the method call `fis.read()` might raise `IOException`. We then instrument this statement as:

```
if (Verify.getBoolean()) i = fis.read();
else throw new IOException();
```

This instrumentation style can precisely mimic the behavior of method `fis.read()`, which has two exclusive consequences:

- The method terminated successfully.
- The method terminated abnormally and an exception is raised.

It is easy to see when the non-deterministic choice `Verify.getBoolean()` returns true, we have the former consequence. When it returns false, we have the latter one. For all the implementation details about program instrumenting, see Appendix A.3.

Figure 3.3 demonstrates an instrumented file whose original file is in Figure 3.1. The newly added line 02

```
import gov.nasa.jpf.jvm.Verify;
```

in the instrumented program introduces the non-deterministic choice API from the back-end model checker JPF. This API is used to trigger exceptions non-deterministically. Line 04 in the original program combines the actions of variables declaration and variable initialization together. Since there is always a probability that during the variable initialization, exceptions might be raised, we have to divide these two actions into two parts in the instrumented

```
01 import java.io.*;
02 import gov.nasa.jpf.jvm.Verify;
03 public class Example1 {
04    public static void main(String[] args) throws IOException {
05        FileInputStream fis;
06        if (Verify.getBoolean()) fis = new FileInputStream(args[0]);
07        else throw new FileNotFoundException();
08        try {
09            int i;
10            if (Verify.getBoolean()) i = fis.read();
11            else throw new IOException();
12            while (i != -1) {
13                System.out.print((char)i);
14                if (Verify.getBoolean()) i = fis.read();
15                else throw new IOException();
16            }
17            if (Verify.getBoolean()) fis.close();
18            else throw new IOException();
19        }
20        catch (IOException ex) {
21            System.err.println(ex);
22        }
23    }
24 }
```

Figure 3.3: Example of Instrumented Program

program. By doing this division, we can have the variable scope working properly while still instrumenting possible exceptions. Line 05 in the instrumented program

```
FileInputStream fis;
```

fulfills the variable declaration task. Line 06–07

```
if (Verify.getBoolean()) fis = new FileInputStream(args[0]);
else throw new FileNotFoundException();
```

in the instrumented program fulfills the variable initialization task. Because the constructor for `FileInputStream` might raise `FileNotFoundException`, this exception is instrumented by using the non-deterministic conditional statement. The same situation happened again in the original program on line 06, where variable declaration `int i` and variable assignment `i = fis.read()` is combined together. Again, our static analyzer breaks it into two parts (line 09 and line 10–11), and instrument the method call `fis.read()` with the potential exception `IOException` (line 10–11).

There are several other program constructs which might raise exceptions. In the original program line 09, method call `fis.read()` might raise `IOException`. This exception is instrumented by the static analyzer as in line 14–15 in the instrumented program.. In original program line 11, method call `fis.close()` might raise `IOException` as well. It is also instrumented as in line 17–18 in the new generated program. In this example, we do not have the case where a single statement which might raise exceptions. All other program constructs such as the class definition, the `try-catch` block and the `while` loop are kept untouched.

Originally, Fex is designed to address all exceptions that can be raised in a Java program, so it supports both checked and unchecked exceptions. By default, we consider all possible exceptions flagged by the static analyzer but also can specify which exceptions to ignore. For example, the unchecked exception, `OutOfMemoryError` could be raised after every new operation and method call. The user might want to turn off the instrumentation of this exception until the more common modes of failure have been addressed.

Fex depends on a configuration file to specify which exceptions are to be ignored. Figure 3.4 gives out an example of a configuration file for verifying program in Figure 3.1. The `<ignore>` item is used to identify these exceptions which are ignored by the static analyzer. `<dir>` specifies the location where the Java file is stored. `<output>` specifies the location where the output file is written. The information is enough for guiding the exception reliability verification, For the future extension of Fex, the configuration file needs to be expanded to provide verification task related information. Details about how the configuration file is used to guide the whole verification process can be found in A.1.

```
<output> /Users/xinli/test/thesisexample/basic-fex/1
<ignore> java.lang.OutOfMemoryError
<ignore> java.lang.NegativeArraySizeException
<ignore> java.lang.ArrayIndexOutOfBoundsException
<ignore> java.lang.ArithmeticException
<ignore> java.lang.NullPointerException
<ignore> java.lang.ArrayStoreException
<ignore> java.lang.ClassCastException
<dir>    /Users/xinli/examples/thesisexample/basic-fex/1/
```

Figure 3.4: Example of the configuration file

3.5 Program Abstraction

A direct application of model checking to a full program of non-trivial size, very quickly leads to state explosion problem. Therefore, it is necessary to reduce the size of the program state as much as possible, while still preserving properties of interest. We deploy a program slicer to serve this end.

A Java program usually contains all kinds of types such as primitive types (`int`, `char`, etc.), class types from Java API and user defined class types from the program itself. For some verification tasks that focus on specific program properties, the type information can give us hints on how to safely abstract the program. For example, when we are conducting exception reliability verification, only the control flow related types such as `Thread` and `Throwable` really matter. Types such as primitive types are actually irrelevant. Therefore, those statements which manipulate only irrelevant type variables have no influence on the program properties we are interested in and can be safely removed. Based on this observation, we may abstract the program by deploying a type-analysis-based program slicing to

transform a Java program into a simplified program with fewer program constructs, data diversities and loop iterations. This simplified program can later be fed into a program model checker for any exception reliability violation.

For exception reliability verification task, we divide Java types into three categories:

1. Control-flow dependent types (CType). Includes program defined classes and all subclasses from class `Throwable` and class `Thread`.

2. Primary types. Such as `Integer`, `char`, etc.

3. Ignored types (LType). All other types besides (a) and (b).

Based on the above type division, we slice the instrumented Java program into a simplified program with several significant changes:

1. Program constructs which involve only ignored types are removed.

   For programming languages like Java, a concrete program is formed by statements and definitions. Statements are the executable code such as assignment, conditions, loops and method calls. Definitions are the declaration code which is used to declare identifiers. For exception reliability checks, we are only interested in if these statements produce any uncaught exceptions. So all types other than control flow related type (CType) are irrelevant here. Therefore, all variable declarations that involve only primitive types and ignored types can be safely removed. For example, the class field definition such as `ServerSocket ss;` can be safely removed but the definition like `IOException ioe;` should be preserved because the latter is a CType (control flow related) variable. For program statements, simple assignments which involve only primitive type and ignored types can removed. For example, an assignment statement such as `i = j + 5;` or statements like `i = fis.read();` can be removed. Conditional and loop statements are treated in a different way in order to preserve the complete control flow, which will be introduced later. For method calls, if the type of the method receiver is an ignored type, they may be removed. For example, in Figure 3.3 line 08, statement `fis.close();` can be removed because the method receiver `fis` belongs to the ignored type. If the method calls are from the user defined classes, they should be preserved because these method calls form the complete control flow.

2. For program constructs which involve not only ignored type variables but also control-flow dependent type variables, these variables with ignored types are filled in with a predefined value.

Program components often have to interact with each other by passing parameters. In real application, providing proper parameters is the basic requirement for the program to behave correctly. However, exhausting all possibilities for these parameters is a great challenge for the model checker and is often impractical. Since we are only focused on exception reliability checking, all exception related information has already be instrumented at the proper place by the static analyzer in the first phase. The value of these parameters with ignored types are irrelevant now. For example, a user defined method

```
byte[] readFile(String filename, int off, int len)
```

takes three parameters: integer `off`, `len` and string `filename`. In the concrete program, the value of these three parameters is essential for the method to function correctly, but now, as we are only interested in exception reliability check, we can use some predefined value to fill in to these parameters just to make the sliced program executable. We use 2 for the integers, and `(String)null` for the string reference types. The sliced program can not fulfill the original functionality, but it still can demonstrate all the exceptional behaviors.

3. The conditional statement whose guard is not program property relevant is now replaced by a non-deterministic choice. All loop constructs are replaced by a fixed iteration loop that executes each loop only a fixed number of times.

   As we are interested in exception reliability verification which is indeed a control flow related program properties, to get the complete program control flow is a challenge. We use the non-deterministic choices to replace these program property irrelevant guard for every conditional statement in order to get the complete control flow. For example, if we encounter a conditional statement such as

   ```
   if (i == 5) ...  ;
   ```

   It is transformed into

   ```
   if (Verify.getBoolean()) ...  ;
   ```

   However, this is not the case for the statement like

   ```
   if (e instanceof IOException) ...
   ```

   because now the guard is program property relevant and should be left untouched. The non-deterministic choice tells the model checker that instead of one specific computation, all possible outcomes of a choice should be considered equally possible.

   For the loop statement, because the exception related information has been already instrumented in the previous process, all we need to do is to guarantee that every statement inside the loop will be executed at least once. Therefore, we transform the loop statement from the style of

`while ( ... ) { ... }`

into a fixed iteration loop as

`for (JPF_index = 0; JPF_index < 2; JPF_index++) { ... }`

to ensure the total coverage of the loop body.

The whole slicing criteria is given out in the form of program transformation rules. For details about these transformation rules and the corresponding program model, see Appendix A.4.

```
01 import java.io.*;
02 import gov.nasa.jpf.jvm.Verify;
03 public class Example1 extends java.lang.Object {
04    public static void main(java.lang.String[] args) throws java.io.IOException {
05        if (Verify.getBoolean()) ;
06        else throw new java.io.FileNotFoundException();
07        try {
08            if (Verify.getBoolean()) ;
09            else throw new java.io.IOException();
10            for (int JPF_index0 = 0; JPF_index0 < 2;JPF_index0++){
11                if (Verify.getBoolean()) ;
12                else throw new java.io.IOException();
13            }
14            if (Verify.getBoolean()) ;
15            else throw new java.io.IOException();
16        }
17        catch (java.io.IOException ex) {
18        }
19    }
20 }
```

Figure 3.5: Example of Sliced Program

Figure 3.5 demonstrate a sliced file whose original file is in Figure 3.3. Line 05 in the instrumented file `FileInputStream fis;` is now removed because this program construct is declaring an object with an ignored type. The same thing happens in line 09 again, where `int i;` has been removed. The code fragment line 06–07 in the instrumented program is used to mimic the behavior of method `fis.read()` where a potential `FileNotFoundException` might be raised. It is now transformed into line 05–06 in the sliced program. Notice that the statement `fis = new FileInputStream(args[0])` has been removed because the receiver of the method call `fis` belongs to an ignored type. For the same reason, the method call `fis.read()`in line 10 and line 14, `fis.close()` in line 17 and `System.err.println()` in line 21 have all been removed. The loop block from line 12 to line 16

`while (i != -1 ) { ... }`

is now transformed into

`for (int JPF_index0 = 0; JPF_index < 2; JPF_index0++ ) { ... }`

Now we only need to execute this loop body twice. Compared to the original program which needs a string argument as the input to function, the sliced program can be model checked

directly. There is no need for providing any arguments.

By program slicing, we now have an abstract program with several significant simplifications:

- In the simplified program, the program constructs that manipulate only primitive types and ignored types are now safely removed. By this statement remove operation, we now have a relatively small program with fewer program state spaces for faster model checking.

- There are some program constructs which are control flow structure dependent. Examples of these program constructs are method parameters and return values. Although these program constructs could be primitive types or ignored types, they are still crucial to achieve the program control flow and therefore should be preserved. However, our program slicer replaces all these primary and ignored type variables with the predefined values (2 for integer, 0 for long and `null` for reference types). By performing this replacement, we greatly reduce the data diversity (we do not need to exhaust all values for certain variables anymore), which reduces the program state space significantly.

- Model checking usually can only addresses the behavior of finite-state closed systems whose future behavior is completely determined by the current state of the system. This is typically done by modelling all possible sequences of events and inputs that can come from the environment. The simplified program is now insensitive to the input from the environment in the sense that all possible exceptions could be raised without any data set. Thus our model checker no longer needs to model the environment anymore.

- As in the simplified program, all loop constructs are replaced by a fixed iteration loop. Because this replacement can minimize the contribution of the loop execution to the system states. It greatly reduces the program state spaces which need to be explored.

3.6 Model Checking

As for the final model checking step, we adopt Java Pathfinder, (JPF) [78], as our back-end model checker. We feed the sliced program into JPF to search for any property violations. JPF is an unusual software model checker in that it directly handles Java byte code. The environment is modelled by non-deterministic choice expressed as method calls to a special class `gov.nasa.jpf.jvm.Verify`. For example, the method `getBoolean()` from class `gov.nasa.jpf.jvm.Verify` returns a boolean value non-deterministically. JPF consists of a custom Java virtual machine that executes the byte code and a search engine that guides

the execution. Since the states–JVM snapshots—are coded into a concise representation, one can use an explicit model checking algorithm to systematically explore all potential execution paths of a program to detect undesired states.

For exception reliability verification, JPF searches for unhandled exceptions. Upon detecting a possible violation, JPF dumps out an execution path leading to the uncaught exception. The execution path can help the programmer to fix the problem. The verification process is repeated until no further violation is reported.

In order to check the exception reliability of the program in Figure 3.1, JPF takes the corresponding sliced program in Figure 3.5 and the configuration file in Figure 3.4 as inputs and generates the error report as in Figure 3.6.[1] From this error report, we can clearly see that the `FileNotFoundException` which is generated at line 06 is unhandled and eventually lead the program into an unreliable situation.

```
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.io.FileNotFoundException
        at Example1.main(Example1.java:6)
====================================================== trace #1
------------------------------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  Example1.java:5                : if (Verify.getBoolean()) ;
------------------------------------------------------ transition #1 thread: 0
gov.nasa.jpf.jvm.BooleanChoiceGenerator[true,>false]
  Example1.java:5                : if (Verify.getBoolean()) ;
  Example1.java:6                : else throw new java.io.FileNotFoundException();
====================================================== snapshot #1
====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.io.FileNotFoundException ..."
```

Figure 3.6: Error report generated by Fex

3.7 Experimental Results

We present experimental results[2] obtained by using our tool on two Java web applications, Project NanoHTTPD [30] and project FizmexWebServer [1]. Table 3.1 shows the running time and state space size of these experiments.

	NanoHTTP	FizmezWebServer
Time	57.6 s	75.3 s
Visited States	15905	38875

Table 3.1: Running time and state space size for the experiments

[1] The statements and line numbers in this error report are based on the sliced program in Figure 3.5, not the original program.

[2] All experiments were performed on a PowerBook with PowerPC G4, 1.67 GHz, 2.0 GB RAM, running Mac OS X 10.4.6, Sun Java SDK build 1.5.0_06-112 and Java Pathfinder Version 3.1.2.

Project NanoHTTPD is a simple embeddable HTTP server application written in Java. After instrumentation and slicing, model checking discovered an uncaught `NumberFormatException`. Examining the execution path that triggered the exception we found that in the main function, there is a call to the library function `Integer.parseInt()`. The fact that this may raise an exception was neglected by the programmer. After fixing this problem, the application has been verified as exception reliable code.

Project FizmezWebServer is an open source Web server application implemented in Java. Applying our tool to this application detected several errors. First, the application has a potential unhandled exception. In method `getServerSocket()`, the socket port number is provided in string format, and therefore needs to be transformed into integer format. The transformation process might throw `NumberFormatException` and is neglected by the programmer. This puts the application into an exception unreliable situation.

3.8 Discussions

We have presented a model checking based verification framework called Fex for verifying exception reliability. Compared to a standard Java compiler, it can detect possible uncaught exceptions without any man-made error-triggering environments. `Fex` can also detect redundant exception handlers.[3]

Every tool for addressing exception reliability problem has two phases. The first phase is used to detect any possible exceptions based on Java specification. This step can only be done by static analysis. Our framework is doing this by reusing Jex's static analyzer. False positives are possible here because we cannot exhaust all data while doing static analysis. For example, for a statement like `a/b`, there is a possibility that an `ArithmeticException` representing divide by zero error might be raised, but if we can guarantee that variable b will never be zero, a false positive alert is inevitable here.

The second phase is used to determine if there are any uncaught exceptions or redundant handlers inside the program based on the program structures and the possible exceptions provided by the first step. This step can be done in several ways. Most of static analysis tools, such as Jex, perform this by using a filtering tool to do the inter-procedural control flow analysis. The filtering tool can calculate the uncaught exceptions based on the formula

$$uncaught = raises + propagates - catches$$

Redundant handlers are caught similarly. which means for a given scope, uncaught exceptions are those raised exceptions from this scope plus the exceptions propagated from the callees minus those exceptions caught by the handlers. On the contrary, Fex is doing this by model checking a program skeleton with only control flows and possible exceptions. Usually,

[3]For details on how this is achieved, please refer to appendix B.

model checking is more precise compared to other static analysis tools in catching runtime program behaviors because the model checker can actually "run" the program. However, since the runtime behaviors for a program with only control flow constructs (*e.g.* exceptions) are statically determined, our model checking based method is no better than these static analyzers for checking exception reliability.

Chapter 4

Extension: a Framework for Verifying Event Sequences

4.1 Introduction

In programming languages, events represent the basic operations of a program. These operations can be method calling, exception handling, locking/unlocking actions etc. The behavior of a software system may depend on the order in which program events are executed.

In practice, these order constraints over program events are called *event sequence related program properties*. Although this kind of property constraints may appear to be overly restrictive, there are many examples of important properties that are expressible in this way. In fact, these event sequence related program properties, once violated, may lead to serious damage. For example, it appears that a Mars lander mission failed in part because the software system designed to assure a soft landing did not deal correctly with the event sequences[55].

The Mars lander software system assumed that when the lander approached the planet, the bump detection variable would be set to false so that the landing gear can be deployed. When contacting with the planet was detected, the bump detection variable will be changed to true which signals the descent engine to turn off. However, in reality, the firing of the lander's retro rockets caused sufficient deceleration to set the bump variable to true. The value of the variable was never checked or reset to false prior to deployment of the landing gear. Once the landing gear was deployed, the condition for shutting down the descent engine was immediately satisfied, and the engine shut down high above the Martian surface. In this software system, the correct event sequence for landing might be: fire retro rockets, reset bump variable to false, deploy landing gear, detect bump, shut down retro rockets. However, the violation of this event sequence causes the loss of the lander.

The situation is even worse when concurrency is introduced. Under concurrent situation,

two runs of a system on identical inputs may produce different results if the order of program events differs for these two runs. Consequently, errors may only manifest themselves under a few possible task schedules. To help detect serious event sequence related program property violations, analysts need tools that can exhaustively consider all possible control flows (including task schedules) and analyze the behavior of the system on each control flow.

In this chapter, we describe our effort to extend basic Fex to address this concern. We choose basic Fex as our start point because of its two merits:

- The basic Fex can get a complete control flow skeleton of the program. This functionality meets the basic requirement for verifying event sequence related program properties. if we can extend our slicing rules to not only preserve all control flow related information such as exceptions and threads, but also preserve those event sequence related program properties we are interested in. The back-end model checker can then examine if there are any violations for these properties.

- Events are generally presented as method calls and it is most often the sequence of method calls that is important in determining event sequence related property. Data, in the form of variables or fields, is generally of secondary importance in determining the set of such sequences and in the specification of valid event sequences. Our program abstraction techniques adopted in basic Fex can be reused here to reduce the state space that needs to be explored by eliminating data diversity. This helps us to make our analysis practical.

With extended Fex , analysts can define a patterns of event behaviors as the specification part, Fex then automatically creates a simplified program that preserves all interesting program properties. Based on the specification and the simplified program, the back-end checking engine JPF determines whether the system satisfies a given property and, if not, provides execution path that lead to the property violation through the simplified program.

4.2 Fex Extension

We extend the basic Fex in two ways:

- We use *executable specification* to specify event sequence related program properties.

- We renew our program slicing rules to preserve not only control flow related program constructs, but also event sequence related program constructs.

4.2.1 Executable Specification for Fex

Previous efforts on specify event sequence related program properties are mainly based on finite state machine (FSM)[9, 28, 71]. A finite state machine $\mathcal{F}$ for specifying event sequence

related program properties is defined as a 5-tuple $\mathcal{F} = (Q, \Sigma, T, q_0, e_0)$ where Q is a finite set called the *states*, representing all possible program states. Σ is a finite set called the *alphabet*, representing interesting program events. $T \subseteq Q \times \Sigma \rightarrow Q$ is a *transition function*, representing the possible transitions between the states. $q_0 \in Q$ is the *start state*. $e_0 \in Q$ is the *error state*. Based on this finite state machine, a practitioner can specify some simple event sequence related program properties such as in which order methods may be called inside a single API.

Figure 4.1 presents us an example of how to use the finite state machine to specify method call sequence constraints for class `java.io.FileInputStream`. In this example, we have four program states: `undefined`, `opened`, `closed` and `error`. All these states form the state set Q. The alphabet set Σ represent the methods we are interested in. In this case, we have four methods namely `new(), open(), close()` and `read()`. q_0 is the start state and in this case, it is the state `undefined`. e_0 is the error state and in this case, it is the state `error`. The transition set T describes the state changing possibilities. This finite state machine starts in state `undefined` where the only legal action is to use `new` operation which initializes a `FileInputStream` object by calling a class constructor. After that, this object is in state `opened` and is ready to execute other operations (`read` or `close`). When calling method `close`, all system resources associated with the `FileInputStream` object is released. Any attempt to execute `read` operation after `close` is not allowed and raises run-time exception.

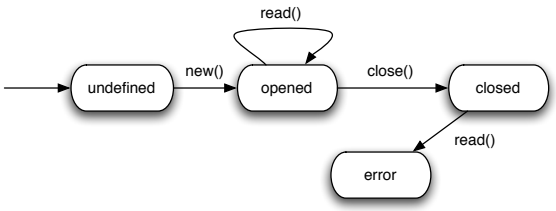

Figure 4.1: An FSM specification of class `FileInputStream`

In order to use Fex to check the event sequence related program properties such as the FSM specification in Figure 4.1, we first need to make all these FSM based properties expressible at the language level. In other words, those states and state changing actions should be presented in an executable program. Also, as the original methods/events contain too much implementation details and introducing too much implementation details into model checking phase can quickly result in state explosion problem, directly model checking event sequence related program properties on the original software is impractical.

We propose to use *executable specifications* to specify the event sequence related program properties. Our executable specifications are written in Java. Each event under investigation

needs to have an executable specification counterpart. The new implementation, while executable, only specifies the event property constraints. All original implementation details are discarded.

With extended Fex, we instrument and slice the user program to get a simplified user program. We then replace the original event implementation with our executable specification. The simplified user program plus the executable specification form our simplified software which can later be fed into the model checker. We replace the original event implementation with the executable specification for two reasons:

- In the simplified user program, all preserved variables with primary types or ignored types are filled with a predefined data value (see Chapter 3). This process reduces the data diversity which is the major contributor to the state explosion problem. However, the original method implementation still need the real value (not the predefined one) to function correctly. Therefore, the original method can not cooperate with the simplified user program. Compared to the original method implementation, our executable specifications are focused on encoding property constraints, the actual data value that the method handles is now irrelevant. For example, the executable specification in Figure 4.2 encodes the property constraints of class `FileInputStream`. It does not care about which specific file is associated with the input stream. Hence, the executable specification can cooperate with the simplified user program on the data diversity reduction.

- As the simplified software (simplified user program + executable specification) supports the data diversity reduction, model checking the simplified software do not need to rely on any data sets. Thus the model checker can skip the environment modeling process.

By model checking the simplified software, we can find out potential event sequence constraint violations.

Using the FSM based specification (Figure 4.1) as an example, the new version of the method implementation is achieved by translating the FSM based program property into an abstract version of these methods implementation. This version of the implementation discards all the details for the real method work. Instead, it only records the state change from the finite state machine specification. By doing this, we not only have the finite state machine specification tractable at the language level, but also have an abstract implementation which hides all the implementation details. Note we only need to give out the executable specification for these methods we are interested in. Other methods inside the same class can be simply neglected by providing an empty implementation. Although this transformation can be done automatically, we are now doing this manually for convenience. The

FSM based program property in Figure 4.1 can be transformed into an implementation as in Figure 4.2.

```
package java.io;

public class FileInputStream extends InputStream {

    private enum State {undefined, opened, closed}
    public State status = State.undefined;
    public final int CONSTANT_INTEGER = 2;

    public FileInputStream(File file) {
        if (status == State.undefined)  status = State.opened;
    }

    public FileInputStream(String s) {
        if (status == State.undefined)  status = State.opened;
    }

    public int read() {
        if (status == State.closed)
            throw new Error ("API Conformance Error! Read after stream closed.");
        if (status == State.opened)  status = State.opened;
        return CONSTANT_INTEGER;
    }

    public int read(byte[] b) {
        if (status == State.closed)
            throw new Error ("API Conformance Error! Read after stream closed.");
        if (status == State.opened)  status = State.opened;
        return CONSTANT_INTEGER;
    }

    public void close() {
        if (status == State.opened) status = State.closed;
    }
    // ... ...
}
```

Figure 4.2: Executable specification for Class `java.io.FileInputStream`

Adopting executable specification gives Fex several advantages:

- Our executable specification is more expressive than FSM based specification (see Section 5.3.1). All FSM based specification can be automatically translated into executable specification. Furthermore, by adopting full Java as our specification language, we can specify the program properties that involve potentially unbounded numbers of objects.

- The executable specification and the program under verification are separated. For a pattern of event behaviors, the corresponding executable specification only need to be prepared once, and can be reused after. Unlike other projects [9, 28] which need extra annotation on the program side, there is no extra annotation burden for Fex.

- Compared to the original implementation, the executable specification is an abstract version which encodes only property constraints. It is a simple event behavior model

of the original implementation with significant less states to be explored. As Fex is based on model checking, this replacement can effectively alleviate the state explosion problem.

There has been a long history of debate on whether or not the specification needs to be executable [44, 36, 40]. Objectors criticize that:

- The demands for high expressiveness and executability are exclusive to each other and the high expressiveness of the specification is always more important, executable specifications should be avoided [44].
- Executable specification inevitably implies abstract implementations. In later implementation, executable specifications can have negative effects because implementers may want to follow the abstract implementation although that may not be desirable [44].

We argue that for our verification task, executable specifications are more practical:

- An executable specification presents a simple behavior model of the system. It is suitable for the model checking method which executes the program and checks the program states exhaustively.
- Safety properties such as "bad things never happen" are natural and efficient to specify by using executable specifications. For example, program assertions are executable and they allow programmers to define safety constraints on program behaviors.
- As we only interested in verifying the event sequence related program property. The event implementation itself is assumed to be correct. There is no negative effect from the executable specification.

The idea of using executable specifications to specify event sequence related program properties is inspired by [70]. They are proposing using a Java like specification language called `EASL/P` to specify the method call constraints. By adopting a subset of Java statements and a restricted set of types (so far only booleans and references), `EASL/P` can handle many event sequence related program properties. We use the whole Java language as our specification language which is more convenient. However, because the executable specification is model checked directly without any abstraction, executable specification need to be carefully written. A too complicated executable specification may cause the state explosion problem.

4.2.2 Program Abstraction Revisited

In basic Fex, all Java types are divided into three categories:

1. Control-flow dependent types (CType). Includes program defined classes and all subclasses from class `Throwable` and class `Thread`.

2. Primitive types.

3. Ignored types (LType). All other types besides (1) and (2).

Based on this type division, we can apply a program slicer that removes all control-flow irrelevant types to get a simplified Java program for model checking exception reliability property. As the exception reliability property only concerns with the program control-flow and our aggressive program slicing procedure can preserve the complete control-flow of the program, the sliced program can be used for exception reliability verification.

However, as we are now interested in event sequence related program properties, blindly removing all control flow irrelevant types will lost all the interesting program properties. In order to solve this problem, we now divide the Java types into four categories:

1. Control-flow dependent types (CType).

2. Primitive types.

3. Crucial types (PType).

4. Ignored types (LType). All other types besides (1), (2) and (3).

Compared to the type category for exception reliability we now have a new category called Crucial types (PType). It includes types which are related to those event sequence related program properties we want to verify.

In exception reliability verification, the program slicer removes these program constructs whose types are control flow irrelevant. The sliced Java program preserves the complete control-flow for model checking. In the updated program slicer, we now treat the Crucial type (PType) as it is a control flow related type. That is, if any program constructs manipulates a PType object, it should be preserved. For example, if we are interested in verifying that certain permission check operation should be performed before the execution of a critical method, then the class which is in charge of the permission check and the class which is in charge of the critical method should all be considered as PType and every event/method comes from these classes should be preserved. Other program constructs which are actually irrelevant to the permission check events can then be safely removed.

4.3 Verification Process Updated

The verification process for event sequence related program properties is very similar to the verification process for exception reliability verification with two major changes:

- We use executable specification to encode designated event sequence related program properties. These executable specification is directly feed into the back-end model checker without any abstraction.
- We are deploying an updated slicer to do the program slicing. We now preserve not only control flow related program constructs but also property relevant (specified by configuration file) objects and events.

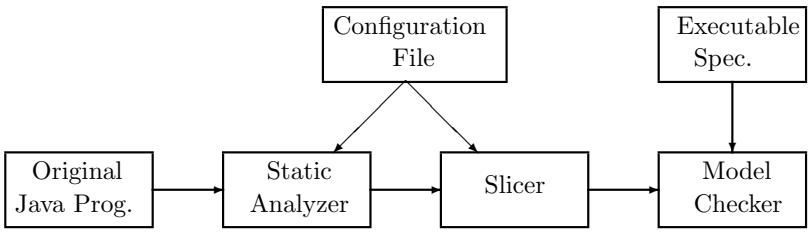

Figure 4.3: Verification Process

The whole verification process is still divided into three steps.

In the first step, we instrument the program so that all possible exceptions can be raised at the proper place. This is the same as in exception reliability verification.

In the second step, we do aggressive slicing for verifying event sequence related program properties. We have an updated program slicing rule set to serve this end. In general, only those program constructs whose types are not control flow related type (CType) or crucial type (PType) are removed. In this way, all interesting events can be preserved. However, as the concrete implementation of these events (they all come from classes with crucial types) are heavily depend on other program constructs which might be eventually removed. The simplified program, although syntactically correct, cannot be executed or model checked. Therefore, it is necessary to use the abstract version of these classes to replace the concrete implementations. Our executable specification actually does this work. For every crucial type class, it should have an executable specification counterpart. For example, if we are interested in verifying if the usage of class `FileInputStream` follows the conformance rules depicted in Figure 4.1, the class `FileInputStream` is then considered as in PType.[1] All corresponding events coming from class `FileInputStream` such as operation `open` and `close` should be preserved. The original implementation of the `FileInputStream` should now be replaced by the executable specification presented in Figure 4.2.

[1]This is done by setting the configuration file. Details are in appendix A.1.

In the third step, we model check the sliced program for any event sequence related program property violations. Since the proper behaviors of these events are actually encoded inside the executable specification, these executable specification need to be directly fed into the model checker along with the simplified program. The executable specification guarantees that every property violation throws an `Error` exception, and the back-end model checker can catch the property violation by detecting these uncaught `Error` exceptions.

The updated verification process for extended Fex is illustrate in Figure 4.3. For detailed examples, please see Chapter 5.

4.4 Limitations

When using abstraction techniques to reduce the number of states of a system, one may introduce more behaviors to be present in the abstract program than in the concrete program, this is called *over-approximation*. Over-approximation can guarantee that if a program property holds in the abstract it also holds in the concrete, but if a program property fails in the abstract then it might not fail in the concrete.

For event based program properties, over-approximation means any event sequences that can occur during an actual execution will be represented by an execution in the abstract program. This is important because it assures that no execution on which a property violation occurs in the concrete program will be overlooked. However, a event sequences violation that occurs in the abstract program may not correspond to executions that could actually occur in the concrete program. In other words, over-approximation does leave the possibility of "false positives", namely identification event sequences violations on the abstracted program that are not actually exist in the concrete program.

Fex adopts several program abstraction techniques that over approximate the concrete program behaviors. It might generate false positives under some conditions:

- Fex might introduce false positives by instrumenting exceptions which may never be raised in the concrete program. For example, in Figure 4.4 line 07, Fex will instrument this statement with the possible `ArithmeticException`.[2] However, as we can see from statement 06, when variable b is equal to zero, the method returns. Therefore, there is never a chance that a `ArithmeticException` will be raised from line 06. So the statement in line 11 will never get executed in the concrete program, but in the abstract program, it will.

- The second way for Fex to introduce false positive is by guard replacement operation. In Fex, every guard of the conditional statement is replaced by an non-deterministic

[2]We assume that in the corresponding configuration file, the `ArithmeticException` is not labelled as to be neglected.

```
01  public static void main(String[] args)
02  {
03     try{
04       int a = ... ;
05       int b = ... ;
06       if (b == 0) return;
07       a/b;
08      ... ...
09     }
10     catch (ArithmeticException ae) {
11           error();
12     }
13  }
```

Figure 4.4: An example for possible false positive introduced by instrumenting impossible exception

choice.[3] However, this aggressive replacement may introduce extra program behaviors. For example, in Figure 4.5, Fex will replace the guards from line 06, 07 and 08 with non-deterministic choices which means line 09 can get a chance to be executed. However, in the concrete program, because the guard in line 08 contradicts with the guards from line 06 and 07, line 09 will never be executed.

```
01  public static void main(String[] args)
02  {
03   long x = ... ;
04   long y = ... ;
05   long z = ... ;
06   if (x<y) {
07      if (y<z) {
08         if (x>z) {
09            error();
10         }
11      }
12   }
13  }
```

Figure 4.5: An example for possible false positive introduced by aggressive guard replacement

Usually, it requires substantial human effort to inspect the output of the model checker to dismiss a false positive. while in our experiment, we have not met such situation. See Chapter 5. Reducing the frequency of false alarms is one of our future goals, for details see Chapter 7.

[3]This replacement will only take place when the guard is not control flow or property relevant. See Appendix A.4 for detail.

The current implementation of Fex supports almost all Java language features including dynamic allocated objects, global variables, concurrency features and exceptions. The only limitation is the handling of arrays. So far, Fex does not support property checking on array elements. That means we cannot handle the situation where the base type of an array is a crucial type. The reason we cannot handle this situation is because we consider the integer type to be property irrelevant which means any program construct involves only integer operation is removed. However, the array manipulation is heavily depend on the integer index operation, if all these index information is removed, we can not differ these elements from each other. Hence in Fex, we cannot handle an array if it is used to store objects coming from crucial types. We could fall back on using a predefined element such as $array[2]$ to present all array elements, but it would introduce false negatives i.e. there a real error in the concrete program but is neglected by Fex. How to handle arrays with crucial types is one of our future work.

4.5 Discussions

event sequence related program property can be either checked statically or at run-time. Event based run-time verification [26] employs dynamic analysis to detect bugs in software during execution. It is concerned with checking a single trace of events generated from the program run against properties described in some logic. When a property is violated, the program can take actions to deal with it. The technique scales since just one model of computation is considered, rather than the entire state space as in model checking. However, runtime verification is akin to program test in the sense that it can never guarantee that a program is error absence. Besides, sometimes when error is detected at run-time, it is too late to do any recovery operations.

In contrast to the run-time verification techniques, there are several techniques which are aiming at verifying the properties statically. Theorem proving relies on the language semantics and a proof system in order to come up with a proof that the program will behave correctly for all possible inputs. Unfortunately, this technique cannot be fully automated for undecidability reasons. Model checking is concerned with checking if all possible traces derived from a program (or its abstract model) satisfy a property of interest. The state-space explosion is known to be an issue when considering concurrency and unbounded types. Additional model abstraction can reduce the model size considerably.

We have presented Fex, a model checking based verication framework which checks the event sequence related program properties for Java program. In contrast to other techniques, Fex creates a simplified program which is relatively small but still conservative with respect to the property that is being evaluated. Fex achieves this reduction by deploying several program abstraction techniques. By applying program abstraction we are trading

the scalability at the cost of precision. In other words, Fex may generate false positives.

As case studies, Fex has been used to address following two event sequence based verification tasks:

- API Conformance Verification
- Java Access Rights Verification

We present API conformance verification in Chapter 5 and Java access right verification is presented in Chapter 6. Our case studies, although preliminary, indicates the effectiveness of our framework. Theoretically, Fex may generate false positives. However, it is surprisingly that in our case studies, no such situation happens.

Although Fex has described as a framework for verifying Java program only. The approach can be expended to handle other byte-code based program languages. Also, In addition to being applicable to programs, the Fex is applicable to other artifacts that capture the flow of events through a system. For example, it could be applied to architectural descriptions or detailed designs.

Chapter 5

API Conformance Verification

5.1 Introduction

Component-based software development is a widely used design approach in the software engineering field. In general, a large system can be decomposed into several functional components. Every component has a well-defined application programming interface (API) which is used for communicating across the components. The API implementers only need to focus on implementing the services that the API promises. As a result, these components can then be reused for other applications. By focus on software reuse, component-based software development greatly improves the efficiency and quality of developing custom applications.

API of a software component declares its methods together with types of parameters and types of results. Whether a method has been passed appropriate parameters is checked at compilation time. However, many APIs impose additional constraints on the order of calling of its methods. For example, a network communication application using the `Socket` API is expected to follow constraints like these:

- Operations such as `getInputStream` or `getOutputStream` can only be applied on a `Socket` object which has already connected to a server.
- After the operation `close`, there should not be any further operations performed on this `Socket` object.

Constraints of this type are called API conformance rules. In general, such rules cannot be checked statically but their violations at run-time raise exceptions. API conformance verification has been studied by several groups with different methodologies such as typestates [28] and abstract interpretation [70]. All these methods have certain constraints. We use the extended Fex tool in order to alleviate the situation.

- We add new rules to the slicer such that we can specify which objects and methods are to be kept as relevant to the API or APIs being verified.

- The API conformance rules are expressed in *executable specifications* which are written as Java classes. These classes form the API implementation at the stage of model checking.

API conformance rules can be divided into two subcategories:

- API usage: there is no nonconforming order of the method calls from a single API or among APIs, see Section 5.3.1.
- Resource usage: all resources are properly released at some point, see Section 5.3.2.

Fex's analysis can guarantee that inside the application, there is no violation of any above kind, see Section 5.3.3.

Fex improves previous work [9, 28, 33, 71] in several ways.

- Our specifications are more expressive. Some previous work [9, 28, 71] on specifying API conformance constraints are based on using finite state machines. Finite state machines are adequate to describe simple behavior inside a single API, but they seem insufficient for describing cooperation among APIs, see Section 5.3. We use the idea of *executable specifications*[70], which can specify all kinds of API constraints.
- In order to verify the resource usage protocol, other works such as the Fugue protocol checker [28] requires not only annotating the corresponding API but also annotating applications as well. Our methods only need to write the executable specification for the method `finalize` from the corresponding API once, see Section 5.3.2.
- To the best of our knowledge, we are the first to consider "all"[1] possible exceptions. This gives us the confidence that our checking results are more accurate than previous work. For example, our tool can eliminate a false positive case reported by [70], which is caused by neglecting a possible exception.
- Previous work [9, 28, 71] performs the verification based on control flow graph whose granularity is too coarse and may miss some subtle errors. Our method is more accurate by adopting the sliced program as program model and using model checking.
- We can handle almost all program constructs, including concurrency. It is well known that finding concurrency related errors is hard for standard static analysis techniques. In fact, all the previous work on API conformance checking known to us cannot handle concurrency. Fex remedies this problem by using the JPF model checker as the checking engine. For example, we can check two multi-thread Java web applications (see Section 5.2) while other tools cannot.

[1]The exceptions we want to handle are configurable and we can explicitly ignore some of them.

- Because previous works [9, 28] are mainly based on typestates (see Section 5.5) that refines object states with finite abstract states, the handling of aliasing becomes a major issue. Typestate based solutions to the aliased objects focuses on making restrictions on the programs or adding extra annotations on the source code. Fex is based on explicit state model checking of Java program, which means we base our analysis on the real virtual machine heap where all aliasing information is coded. Therefore, we don't need any extra annotations or restrictions on the source code to do the analysis, as the back-end explicit state model checker can handle all of this.

The rest of this chapter is organized as follows: in Section 2 we use a simple example to illustrate an API conformance problem and how the Fex tool is used to verify it. In Section 5.3 we introduce Fex's specifications for the API conformance constraints. Section 5.3.3 introduces the verification process, and Section 5.4 presents the experimental results. Section 5.5 summarizes related work.

5.2 Motivating Example

The event sequence constraints described in Figure 4.1 actually presents a simplified API conformance specification for `java.io.FileInputStream`. We would like to verify that a given program does not violate `FileInputStream` conformance rules statically. Instead of running the actual program, we use the JPF model checker which examines all execution paths. The idea is to replace the concrete implementation of `FileInputStream` with an executable specification of `FileInputStream` conformance rules. Please note that the executable specification of conformance rules is a much simpler program than the actual implementation which implicitly encodes the rules.

Figure 4.2 presents the executable specification of `java.io.FileInputStream` implementing the FSM from Figure 4.1. The FSM state is represented by the field `status`. Transition relations are coded as `if` statements. For example, the `if` statement inside method `close` changes the program state from `opened` to `closed`. When the FSM enters the `error` state, an exception is raised in the program, representing a possible conformance violation.

We would like to examine that a very simple program in Figure 5.1 obeys the conformance rules of `FileInputStream`. The program first calls the method `initialize` which creates a `FileInputStream` object `is` connected to file named `args[0]`, and a `FileOutputStream` object `os` connected to file named `args[1]`. The method `copy` copies the contents from `is` to `os`. Method `cleanUp` is called when an exception situation is raised.

The original program in Figure 5.1 is converted into a sliced Java program as in Figure

```
03 class Test {
04    private FileInputStream is;
05    private FileOutputStream os;
06    void initialize(String s1, String s2) {
07        try{
08            is = new FileInputStream(s1);
09            os = new FileOutputStream(s2);
10        }
11        catch (IOException e) {
12            System.out.println("Catching Exceptions, now clean up.");
13            cleanUp();
14        }
15    }
16    void copy () {
17        try{
18            int i = is.read();
19            while(i != -1) {
20                os.write(i);
21            i = is.read();
22            }
23        }
24        catch (IOException e) {
25            System.out.println("Catching Exceptions, now clean up.");
26            cleanUp();
27        }
28    }
29    void cleanUp() {
30        try{
31            if (is != null) is.close();
32            if (os != null) os.close();
33        }
34        catch (IOException e) {
35            // do something ...
36        }
37    }
38    public static void main (String[] args) {
39        Test t = new Test();
40        t.initialize(args[0], args[1]);
41        t.copy();
42    }
43 }
```

Figure 5.1: Example program using `java.io.FileInputStream`

5.2[2]. We perform two actions during the conversion (see Chapter 3).

- We instrument the program so that the program can raise all possible exceptions non-deterministically.
- We slice the program in order to make the program smaller.

Both actions are controlled by a configuration file as in Figure 5.9. This file tells the converter what kind of exceptions should be instrumented and which APIs are considered relevant and should be preserved. In this example, because we mentioned both `FileInputStream` and `FileOutputStream` in the configuration file, objects `is` and `os` with corresponding operations are all preserved. At the model checking phase, the model checker examines the sliced program together with the executable specification for API `java.io.FileInputStream` (Figure 4.2). If there is any conformance rule violation, the model checker generates a report which points out the erroneous execution track.

Figure 5.3 presents the report Fex generated. The report starts with the stack trace when the program stopped due to uncaught exception

```
API Conformance Error! Read after stream closed.
```

This exception is thrown by our executable specification. The following part presents the step by step execution trace which leads to the error (non-deterministic choices are documented). The execution trace tells us that an `FileNotFoundException` was thrown at line 12 and was caught at line 14. Method `cleanUp` was called in turn at line 15 which closed the input stream `is`. Then, method `copy` (line 51) was executed which attempted to call operation `read` (line 20) on closed stream `is`. It violates the API conformance rule of `FileInputStream`.

This error is triggered by calling the `cleanUp` too early inside `initialize`. Usually the clean up method is called at the end of the whole action (preferably in a `finally` block) so that there is no side effects. However, in this example, trying to do the cleaning in method `initialize` results in a closed stream `is` which is later referenced. For this example, the proper way would be to add a `try-finally` block in method `main` and call `cleanUp` in the `finally` section.

Please note that there is a special error which is not explicitly represented in the FSM specification. In the state `undefined`, the only legal operation is to execute a class constructor. Attempting any other operation results in a `NullPointerException` . The above program can lead to this situation. If operation `new FileInputStream()` on line 08 fails, variable `is` is null and an `IOException` is raised. Continuing to execute `cleanUp` at line 13 is OK, but executing method `copy` which attempts operation `read` on an null object `is` raises `NullPointerException`. Fex catches this error as well.

[2]This automatically generated sliced program has been edited due to space constraints.

```
04 class Test extends java.lang.Object {
05    private java.io.FileInputStream is;
06    private java.io.FileOutputStream os;
07    void initialize(java.lang.String s1, java.lang.String s2) {
08      try {
09          if (Verify.getBoolean()) is = new FileInputStream((java.lang.String)null);
10          else throw new java.io.FileNotFoundException();
11          if (Verify.getBoolean()) os = new FileOutputStream((java.lang.String)null);
12          else throw new java.io.FileNotFoundException();
13      }
14      catch (java.io.IOException e) {
15          this.cleanUp();
16      }
17    }
18    void copy() {
19      try {
20          if (Verify.getBoolean()) is.read();
21          else throw new java.io.IOException();
22          for (int JPF_index0 = 0; JPF_index0 < 2; JPF_index0++){
23              if (Verify.getBoolean()) this.os.write(2);
24              else throw new java.io.IOException();
25              if (Verify.getBoolean()) this.is.read();
26              else throw new java.io.IOException();
27          }
28      }
29      catch (java.io.IOException e) {
30          this.cleanUp();
31      }
32    }
33    void cleanUp() {
34      try {
35          if (is != null) {
36              if (Verify.getBoolean()) is.close();
37              else throw new java.io.IOException();
38          }
39          if (os != null) {
40              if (Verify.getBoolean()) os.close();
41              else throw new java.io.IOException();
42          }
43      }
44      catch (java.io.IOException e) {
45          // do something ... ...
46      }
47    }
48    public static void main(java.lang.String[] args) {
49        Test t = new Test();
50        t.initialize((java.lang.String)null, (java.lang.String)null);
51        t.copy();
52    }
53 }
```

Figure 5.2: Sliced program based on Figure 5.1

```
... ...
==================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.Error: API Conformance Error! Read after stream closed.
        at java.io.FileInputStream.read(FileInputStream.java:19)
        at Test.copy(Test.java:20)
        at Test.main(Test.java:51)
==================================================== trace #1
---------------------------------------------------- transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  Test.java:49     : Test t = new Test();
  Test.java:50     : t.initialize((java.lang.String)null, (java.lang.String)null);
  Test.java:9      : if (Verify.getBoolean()) is = new FileInputStream((java.lang.String)null);
---------------------------------------------------- transition #1 thread: 0
gov.nasa.jpf.jvm.BooleanChoiceGenerator[>true,false]
  Test.java:9      : if (Verify.getBoolean()) is = new FileInputStream((java.lang.String)null);
  Test.java:11     : if (Verify.getBoolean()) os = new FileOutputStream((java.lang.String)null);
---------------------------------------------------- transition #2 thread: 0
gov.nasa.jpf.jvm.BooleanChoiceGenerator[true,>false]
  Test.java:11     : if (Verify.getBoolean()) os = new FileOutputStream((java.lang.String)null);
  Test.java:12     : else throw new java.io.FileNotFoundException();
  Test.java:14     : catch (java.io.IOException e) {
  Test.java:15     : this.cleanUp();
  Test.java:35     : if (is != null) {
  Test.java:36     : if (Verify.getBoolean()) is.close();
---------------------------------------------------- transition #3 thread: 0
gov.nasa.jpf.jvm.BooleanChoiceGenerator[>true,false]
  Test.java:36     : if (Verify.getBoolean()) is.close();
  Test.java:39     : if (os != null) {
  Test.java:46     : }
  Test.java:47     : }
  Test.java:17     : }
  Test.java:51     : t.copy();
  Test.java:20     : if (Verify.getBoolean()) is.read();
---------------------------------------------------- transition #4 thread: 0
gov.nasa.jpf.jvm.BooleanChoiceGenerator[>true,false]
  Test.java:20     : if (Verify.getBoolean()) is.read();
==================================================== snapshot #1
... ...
```

Figure 5.3: Error message generated by Fex

5.3 Fex for API Conformance Verification

Usually a software can be divided into two parts: user programs and the API libraries. API makes it easier to develop a program by providing all the building blocks, a user put these blocks together in the user program to perform certain functionality. API blocks (components) interact with each other in the user program, if there is any error with in the interaction, a run-time exception is raised.

API conformance rules regulate the order in which API methods may be called. These rules are implicitly encoded inside the actual implementation. Runtime exceptions are raised when these conformance rules are violated. The descriptions of these rules are written as comments to the API and are organized in an informal and unstructured way. In order to do the API conformance verification, formal specification of these rules is required.

5.3.1 Specifying API Usage Protocol

Some previous efforts [9, 28, 33, 71] on formally specifying API conformance rules are based on finite state machine. Fugue [28] specifies the FSM specification using program annotations. Figure 5.14 presents an annotated API example in Fugue. The annotated `TypeStates` forms the program states. All methods form the alphabet set. Transition function is defined as a pre/post condition pair of a method. [9] is an improvement on Fugue which inherit the annotation style FSM specification. Project CHET [71] uses the finite state machine specification where predefined program events form the program states. Those events represent program actions such as method call or field access. [33] also uses an FSM based specification.

Finite state machines are adequate to describe simple behavior inside a single API, but they are insufficient for handling unbounded data and are inconvenient when data values get bigger. We propose to use *executable specifications* to specify the API conformance rules. Our executable specifications are written in Java. Each API under verification needs to have an executable specification Java class. The new class while executable only specifies the API conformance rules. For example, our executable specification for `java.io.FileInputStream` is implemented as in Figure 4.2.

Our executable specification is more expressive than FSM based specification. All FSM based specification can be automatically translated into executable specification. Furthermore, by adopting full Java as our specification language, we can specify the conformance situation that involve potentially unbounded numbers of objects such as API cooperation issues.

We use the Java `Collection` API and the corresponding *Concurrent Modification Problem* (CMP) as an example to illustrate the API cooperation issue. Every Java collection set has an iterator that is used to iterate over all elements inside the collection. And, in order to ensure that the inner structure of the collection does not collapse at runtime, there is an implicit constraint for the correct usage of the iterator. Once an iterator is created on a collection, the only valid modification one can conduct on that collection is through the iterator itself. There are usually two ways to violate this constraint.

- By modifying the collection after the creation of the iterator.
- By creating a new iterator on the collection and doing the modification through the new iterator.

Both operations lead to inconsistent iterator states. Once these inconsistent situations are detected at runtime, a `ConcurrentModificationException` is thrown. As the research shows that this concurrent modification exception is often related to subtle program errors [70], how to statically determine if all uses of iterators are valid is a challenge.

```
public class Version {
}
public class JPF_Collection implements Collection {
    public Version ver;

    public JPF_Collection() {
        ver = new Version();
    }

   public boolean add(Object i) {
        ver = new Version();
        return Verify.getBoolean();
   }

     public Iterator iterator() {
         return new JPF_Iterator(this);
     }

    public boolean remove(Object i) {
        ver = new Version();
        return true;
    }
// ... ...
}
public class JPF_Iterator implements Iterator {
    JPF_Collection col;
    Version defVer;

    public JPF_Iterator(JPF_Collection c) {
        defVer = c.ver;
        col = c;
    }
    public boolean hasNext () {
        return Verify.getBoolean();
    }

    public Object next() {
        assert defVer == col.ver :
        "---Concurrent Modification Problem Raises!---" ;
        return this;
    }
    public void remove() {
        assert defVer == col.ver :
        "---Concurrent Modification Problem Raises!---" ;
        col.ver= new Version();
        defVer = col.ver;
    }
// ... ...
}
```

Figure 5.4: Executable specification: Class `JPF_Iterator` for representing interface `java.util.Iterator` and Class `JPF_Collection` for `java.util.Collection`

The class `Collection` and `Iterator` in Java are both interfaces providing descriptions for common data container and corresponding iteration operations. These two interfaces cannot be instantiated and do not provide executable methods. Every program using `Collection` or `Iterator` must provide concrete Java classes to implement them. These programs is expected to obey conformance rules for the `Collection` and `Iterator` as described above. We are interested in verifying that a given program does not violate these conformance rules.

Based on `EASL/P`'s description of the CMP problem [70], we implement two concrete classes `JPF_Iterator` and class `JPF_Collection` as the executable specifications for `Collection` and `Iterator`. presented in Figure 5.4.

From this specification, we can see that every time a `Collection` object updates (through method `add` or `remove`), it's `Version` field gets a new instance. When executing method `Iterator::remove` or `Iterator::next`, the `Version` field of the `Iterator` object is compared with the `Version` field of the back-end `Collection` object. If the comparison result is negative, a concurrent modification problem is raised. As class `Collection` can be updated many times, we need to express behaviors among potentially unbounded numbers of `Version` instance. Hence, finite state machine can not specify CMP problem.

An alternative way to achieve this executable specification is to write an abstract implementation by inspecting the real Java library implementations for `Iterator` and `Collection`, which uses the arithmetic operations to do the version check.

5.3.2 Specifying Resource Usage Protocol

In [28], resources are defined as objects which need to be released by a certain method call rather than by garbage collection. An instance of an API is considered as resource object when that API has a clean up method fulfilling the release action. A resource usage protocol simply stipulates that every resource object should be released once it is no longer used. In other words, for a resource object, the clean up method has to be executed before the object is garbage collected. This is indeed an API conformance rule.

In the previous section, we only introduce how to extend Fex to check nonconforming method calls, we now focused on how to extend Fex to check resource usage protocols.

Resource usage has a practical relationship with the exception handling. We use an example to illustrate how exceptions may cause incorrect resource usage. Figure 5.5 presents the sample Java program. This program first tries to acquire two resources: `socket` and `dataOutputStream`, then performs some operations. When leaving the `try`, the program attempts to release these two resources. At the first glance, it should work properly since all the `close()` operations are inside a `finally` block. however, the code is flawed:

- A failure to close the `dos` object (Figure 5.5 line 12) will raise an exception and lead

to line 13 `socket.close()` being bypassed. Therefore, resource `socket` leaks.

- If operation `serverSocket.accept()` on line 4 or `new DataOutputStream()` on line 5 fails, variable `socket` or `dos` will be null on line 12 or 13. Then a null pointer dereferencing error will occur.

```
01 Socket socket = null;
02 DataOutputStream dos = null;
03 try {
04      socket = serverSocket.accept();
05      dos = new DataOutputStream(socket.getOutputStream());
06      // other operations
07     }
08 catch (IOException e) {
09      // handling exceptions here
10      }
11 finally {
12          dos.close();
13          socket.close();
14         }
```

Figure 5.5: Example of Original Program

First Attempt: Generating Specifications for Resource Leaks

Our first attempt [59] was to use automatically generated resource usage related annotations to support the verification. This method is similar to Fugue protocol checker (see Section 5.5), only our annotation is generated automatically.

```
public class Resource {
... ...
    public boolean inuse = true;

    public void open() {
        inuse = true;
    }

    public void close() {
        inuse = false;
    }
... ...
}
```

Figure 5.6: Resource Class Example

Compared to basic Fex, which uses three steps for exception reliability verification (Chapter 3), we now need four steps.

1. Instrumentation. This process is the same as with reliability checking.

2. Slicing. This process is the same as with reliability checking except that now we preserve not only exceptions and control flow constructs but also all resource usage related information.

3. Annotation. We use a generic `Resource` class (see Figure 5.6) to replace all those classes which are considered as resources. Since the resource usage protocol requires no resource leaking guarantee, we need to translate the implicit specification "no resource leak" into assertions and annotate these assertions at the proper places. In that way, a model checker can report any resource leak flaws in the fourth (model checking) step. For example, Figure 5.7 presents the produced program after instrumentation, slicing and annotation. The original program is in Figure 5.5. For the details about how this is achieved, see below.

4. Model checking. This process is the same as with reliability checking.

```
01  Resource socket= null;
02  Resource dos= null;
03  try {
04    socket = null;
05    dos = null;
06      try {
07           if (Verify.getBoolean()) socket = new Resource();
08              else socket = null;
09           if (Verify.getBoolean()) throw new java.io.IOException();
10           if (Verify.getBoolean()) dos = new  Resource();
11                else dos = null;
12           if (Verify.getBoolean()) throw new java.io.IOException();
13           // other operations
14           }
15      catch (java.io.IOException e) {
16              // handling exceptions here
17            }
18      finally {
19              dos.close();
20              if (Verify.getBoolean()) throw new java.io.IOException();
21              socket.close();
22              if (Verify.getBoolean()) throw new java.io.IOException();
23           }
24      }
25  finally {
26  assert socket == null || socket.inuse == false;
27  assert dos == null || dos.inuse == false;
28  }
```

Figure 5.7: Example of Sliced & Annotated Program

In order to do the resource usage check, specification annotations need to be inserted as assertions into the sliced Java program. JPF then examines all the execution paths, and if along any path there is an assertion violation, we have a resource leak.

Program resources may be declared at two levels: method and class. At the method level, resources are declared as local objects that should be released before termination of the method. (Note: this only works for the situation where the return type is not a resource type.) For each method, our annotator automatically inserts a new `try-finally` block. The new `try` block encloses the original method body while assertions that all resources have been released are inserted into the new `finally` block. In that way, JPF can check if

there are any assertion violations at the end of method execution. In order to observe scope issues, all method level resource objects should be predefined before the new `try-finally` block. This work can be automatically done by the annotator.

At the class level, it is usually difficult to decide the exact point where a resource should be released. Additional information about the interactions between classes is required, and users may need to insert additional specifications. However, for transaction style applications (such as web applications), the objects that handle a transaction have a well-defined life cycle. When a class is a descendant of `Thread` or `Runnable`, all related external resources should be released before the `run` method terminates. Therefore, in this particular case, we can automatically annotate assertions to the `run` method just as for ordinary methods.

Second Attempt: Extending Fex Specification

In our first attempt, we check resource usage protocols by inserting some automatically generated annotations. However, it still has two restrictions.

1. We cannot handle situations where the global variables and method return values are resource types (in these cases, we are not sure where these resources should be released).

2. This method needs extra annotations. The protocol checker Fugue from Microsoft Research [28] suffers the same problem.

In this section, we introduce our second attempt to use the executable specification to specify the resource usage protocol which can avoid these restrictions.

Looking through our first attempt, almost all information about the resource is provided. We know which objects are considered as resources (specified by the configuration file). We know how these resources are manipulated through the program execution (with the help from executable specification and the model checker). The only reason that we need extra annotations is because we need to notify the checker to check if the resource is released at a proper place. Theoretically, this is a liveness problem, stating that "good things (resources released) will eventually happen at some points". As we can imagine that every object dies sometime, if we can spot that time slot, it would be an ideal place to check if the resource object is released.

In object oriented programming languages, every object has a well defined life cycle while the memory management mechanism is different from one to another. C++ uses a programmer governed mechanism. Every class in C++ has an destructor and it will be called when the object dies. On the other hand, languages like Java use an automatic garbage collector that will reclaim objects once it figures out the object is no longer used. Although Java does not have an explicit class destructor, it provides the `finalize()` method to

```
public class FileInputStream extends InputStream {
  // ... ...
    public void close() {
        if (status == State.opened) status = State.closed;
    }

    protected void finalize() {
        if (status != State.closed) throw new Error("---FileInputStream leak!---");
    }
  // ... ...
}
```

Figure 5.8: Adding `finalize` as specification for `java.io.InputStream`

provide clean up action support. Java specification states that every time when an object is about to be garbage collected, its corresponding `finalize` method will be called. Therefore, to check if a resource object has been released or not at its `finalize` method would be a good choice.

There are several technical problems.

- Although Java specification guarantees that every time when an object is garbage collected, its `finalize` methods will be executed. It does not regulate when the garbage collector is called. That means there is a chance that although an object is no longer used, the garbage collector is never called and therefore the `finalize` is never executed (for example, a situation where you have a lot of memory and run a very small Java program). However, since we are doing model checking, the model checker can guarantee that, at least when the program terminates, the garbage collector will be executed once. In fact, our model checker runs the garbage collector more frequently than that. Since the model checker needs to eliminate dead objects constantly in order to keep a relatively small heap to check.

- Technically, every resource release method may fail, causes runtime exception. That means every application has a tiny chance to leak resource. As we are only verifying that the programmer did not forget to release resource, whether the release operation success or not is out of the verification scope. Therefore, we instrument every resource release method in a way that the method is guaranteed to be successful. For example, instead of instrument `FileInputStream` method `close` in a traditional way as

  ```
  if (Verify.getBoolean()) is.close();
  else throw new java.io.IOException();
  ```

  We instrument it as

  ```
  is.close();
  if (Verify.getBoolean()) throw new java.io.IOException();
  ```

- The original version of the model checker JPF does not support the `finalize` method due to performance considerations. We modified the model checker to support `finalize` and found out the performance penalty was negligible.

Based on this idea, we extended the Fex executable specification to support the `finalize` method. Inside that `finalize` method, we check if the object is released. If not, we throw an exception notifying that there is a resource leak. For example, we add the `finalize` method to the class `java.io.FileInputStream` as shown in Figure 5.8. Every time an `FileInputStream` type of object dies, the `finalize` method is executed and if that object is not in the `closed` state, an exception is generated which will later be caught by the model checker.

5.3.3 Verification Process

Compared to basic Fex (Chapter 3), which is used for exception reliability verification, the verification process for API conformance is almost the same, with two changes:

- We use our executable specification which encodes the API conformance rules to replace the original API implementation.
- We are deploying an updated slicer to do the program slicing. We now preserve not only control flow related program constructs but also relevant (specified by configuration file) API objects and operations.

The executable specification guarantees that every nonconforming behaviors throws an `Error` exception, a model checker can catch the API conformance error by detect this uncaught `Error` exception.

We use the example program from 5.1 to illustrate the API conformance verification process.

- First, we perform the instrumentation. The instrumentation step is guided by the configuration file. Compared to the configuration file for exception reliability checking, we have a new item `<crucial>` here. This item is used to specify those API classes which are relevant to the verification task. For example, the configuration file presented in Figure 5.9 is for the program in Figure 5.1. Here, the APIs which are relevant to our job are `java.io.FileInputStream` and `FileOutputStream`.
- Second, we perform the program abstraction. We apply a program slicer to remove all program constructs that are irrelevant for checking API conformance. The sliced program contains nothing but the control flow constructs and all verification related API class objects and operations. Figure 5.2 presents the sliced program.

```
... ...
<ignore> java.lang.ArrayStoreException
<ignore> java.lang.ClassCastException
... ...
<dir> /Users/xinli/examples
<crucial> java.io.FileInputStream
<crucial> java.io.FileOutputStream
```

Figure 5.9: Example of configuration file for API conformance verification

- Finally, we feed the sliced program into the software model checker, Java Pathfinder (JPF) [78]. It checks if there are any uncaught API conformance related exceptions in the sliced program. If so, JPF dumps out an execution path leading to the violation.[3]

5.4 Experimental Results

We present experimental results obtained by applying extended Fex on several API conformance verification tasks. These tasks can be divided into three categories, we introduce them one by one.

5.4.1 Concurrent Modification Problem

For *Concurrent Modification Problem* (CMP), we need to verify that the state of the iterator and the back-end collections are consistent (see section 5.3 for the detail). Since neither API `Iterator` nor `Collection` has an explicit object release method, we do not need to do the resource usage check here.

We use the benchmarks from [70] as our test cases. These benchmarks are deliberately designed as a "stress test" for CMP problem. They include the typical erroneous usage of the iterators and collections in the real application, such as using the old iterator after the collection has been modified. and using the old iterator while the aliased iterator has done modification. These benchmarks also inspect various difficult aspects of CMP problem such as aliasing, exceptions, inter-procedural call and complex data structure where collections are deeply embedded inside.

We can find all the CMP errors (12 in total) that the reference [70] could find. However, the reference [70] does report one false positive from these benchmarks. The reason for this false positive is due to the limit support for exceptions. Since our tool has a full support for exceptions, our tool does not generate any false positive.

Compared to previous work in [70], our methodology also has the advantage to handle some complicated cases. for example, for the case in Figure 5.10, there is a potential concurrent modification problem in line 05 which is caused by the recursive invocation of method P. Previous work often fails to detect this error while our tool can.

[3]For details, please see Appendix A.5.

```
01 void P (Set s) {
02   Iterator i = s.iterator();
03   if (...) {
04      P(S);
05      i.next();
06   } else {
07     i.next();
08     i.remove();
09     i.next();
10   }
11 }
```

Figure 5.10: a complicate situation in `Iterator` case

5.4.2 Socket API and Java IO Stream API

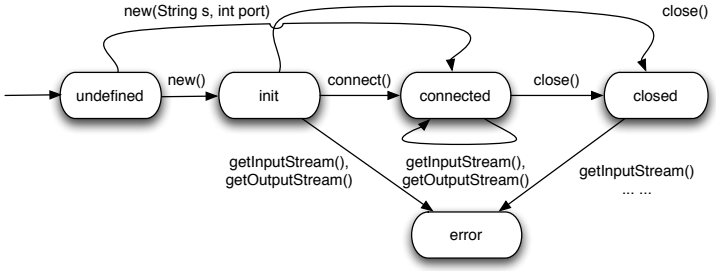

Figure 5.11: API specification for API `java.net.Socket` in FSM

We verify the `java.io.Socket` API and several `java.io.*` APIs. the conformance properties we want to verify are:

- API usage: the order of the API method calls agree with the specification described in Figure 4.1[4] and 5.11.
- Resource usage: all resources are released at a proper place.

The benchmarks we use in this verification task include two multi-thread Java web applications namely project NanoHTTPD [30] and project FizmezWebServer [1]. The source code can be downloaded from their websites.

API usage protocol

We did not find any API usage violation in these two applications.

Resource usage protocol

for the resource usage protocols, we found 6 resource leak problems in total. 4 of them are at the method level. In project NanoHTTPD, A `BufferedReader` type object `in` is declared

[4]Other IO stream API specification such as `DataOutputStream` and `BufferedReader` are very similar to this one.

in method `run()` in class `HTTPSession` and a `FileInputStream` type object `fis` is declared in method `serveFile()` in class `NanoHTTPD`. The author closes these resources at the end of the corresponding try block but not in a finally section. As a result, when exceptions are thrown these resources are never closed and thus may leak. The same situation also happens in the project FizmezWebServer. Inside method `getConfig()`,although the programmer comments the `bufferedReader.close()` saying that "close the mime file so others can use it!" there is still a resource leak problem with the object `bufferedReader` because it is not protected by a finally block. The same problem happens again in method `getMimeTypes()` in FizmezWebServer, leading to a `BufferedReader` type of object `execBufferedReader` not being properly released.

There are also two subtle class level resource leak problems. First, the object `mySocket` of type `java.net.Socket` is defined as a field in class `NanoHTTPD`. It is initialized when an `HTTPSession` is created and should be closed before this HTTP session is terminated, that is before the corresponding run method ends. The program does call `mySocket.close()` by the end of the `run()` procedure, but this call may be not executed as it is not enclosed in a finally block. JPF shows us an execution path which under high load causes an exception prior to the socket closure, and leads to the `mySocket` leak. Thus the application can fail under a very high number of requests. Another class level error lies in the project FizmezWebServer. a `java.net.Socket` type of object `socket` is defined as a field in class `WebServer`. The author intends to close it by the end of the try block in run method. But since there are many operations which might throw exceptions inside the try prior to the close operation, we find yet another leak.

5.4.3 API java.sql.*

API package `java.sql.*` from Java provides the API for accessing and processing data stored in a database. When used, it should obey the following constraints: client programs can create `Connection` to databases. Any number of `Statements` can be created over a `Connection`. A `Statement` can be used to execute an SQL query over the database by using `executeQuery()` method, which returns the results to the query as a `ResultSet`. The `next` method from class `ResultSet` is used to repeatedly iterate over the results of the query. However, the execution of the `executeQuery` method of a `Statement` implicitly closes any `ResultSet` previously returned by the `Statement`. Therefore, it is illegal to use any of these `ResultSet` any more. Similarly, after closing a `Connection`, it is illegal to use any of the `Statements` created from that `Connection` or any of the `ResultSet` returned by these `Statement`. These constraints can be describe using our executable specification as in Figure 5.12.

```
public class JPF_Connection implements java.sql.Connection {
  boolean closed;
  HashSet statements;
  public JPF_Statement createStatement() {
    assert !closed;
    JPF_Statement st = new JPF_Statement(this);
    statements.add(st);
    return st;
  }
  public void close() {
    closed = true;
    Iterator its = statements.iterator();
    while (its.hasNext()) {
       JPF_Statement s = (JPF_Statement)its.next();
       if (s.myResultSet != null) {
           s.closed = true;
           s.myResultSet.closed = true;
       }
    }
  }
  // ... ...
}

public class JPF_Statement implements Statement {
  boolean closed;
  JPF_ResultSet myResultSet;
  JPF_Connection myConnection;
  public JPF_Statement (JPF_Connection c) {
    closed = false;
    myConnection = c;
    myResultSet = null;
  }
  public JPF_ResultSet executeQuery(String qry) {
      assert !closed;
      if(myResultSet != null) myResultSet.closed = true;
      myResultSet = new JPF_ResultSet(this);
      return myResultSet;
  }
  public void close() {
      closed = true;
      if (myResultSet != null) myResultSet.closed = true;
  }
  \\ ... ...
}

public class JPF_ResultSet implements ResultSet {
  boolean closed;
  JPF_Statement ownerStmt;
  public JPF_ResultSet (JPF_Statement s) {
    closed = false;
    ownerStmt = s;
  }
  public void close() {
    closed = true;
  }
  public boolean next() {
    assert !closed;
    return Verify.getBoolean();
  }
  // ... ...
}
```

Figure 5.12: The executable specification for Java JDBC assessment

consider the Java program fragment from Figure 5.13 [5]. This program performs a number of database manipulation operations. There is a subtle error which violated the API conformance constraints we presented in the last paragraph. The buggy scenario is as follows: the execution of a query in line 28, which executing a database query on `stmt2`, has an unnoticed side effort of discarding the results to the previous query executed in line 27 (because they are executing the query on the same `Statement`). Therefore, the subsequent attempt to use the discarded result `rs2` in line 40 is invalid.

As this JDBC manipulation constraint is a typical example of API conformance check. We apply our extended `Fex` to solve this problem. We use several benchmarks from [81] as our test cases. The first test case named `JDBCExample` is an extended version of the running example that uses several `connection`s and `Statement`s. There is an deliberately implanted erroneous scenario as follows: The program closes a database `Connection` under a very rare exception situation. However, another `Statements` is still associated with this `Connection` and is dereferenced after the closing operation of the `Connection`. Therefore, an illegal dereferencing problem raises. Our framework can quickly detect this erroneous situation.

We also use a relatively big example named `SQLExecutor` as our test case. `SQLExecutor` is an open source JDBC framework based on `java.sql.*` API. The whole implementation of `SQLExecutor` is about 2000 lines of `Java` code. As previous effort such as [33] fails to verify this example due to the complex program relations, our framework can have this program successfully verified in about 500 seconds. Our framework reports that there is no possible conformance violation inside this JDBC framework.

5.5 Related Works

There are several projects focus on API conformance verification. The project that is closest to our work is Fugue [28]. Fugue is a static software analyzer based on typestate checking. Typestates [76] specify extra abstract states of objects beyond the usual types. Operations which change object states change their type state as well. Fugue uses extra annotations to specify API conformance protocols and aliasing information. For conformance protocol, Fugue need to annotate the API's source code to describe the typestates and the state changing properties (the pre typestates as preconditions and the post typestates as the postconditions). Sometimes it also needs to annotate on the applications to describe the resource usage protocol and typestates. For aliasing information, every object need to be labelled as either `NotAliased` or `MaybeAliased`. `NotAliased` means the object has only one unique pointer reference. `MaybeAliased` means there might be many pointers refer to this object. By applying aliasing rules such as "a `NotAliased` parameter can only be assigned

[5] This example is borrowed from [81].

```
10  ConnectionManager cm = new ConnectionManager();
11  Connection con1 = cm.getConnection();
12  Statement stmt1 = cm.createStatement(con1);
    // ... ...
15  ResultSet MaxRs = stmt1.executeQuery(maxQry);
16  if (maxRs.next())
    // .... ...
18  ResultSet rs1 = stmt1.executeQuery(balanceQry);
19  if (maxBalance1 < threshold) {
20  stmt1.close();
21  closed1 = true;
22  }
23  Connectionn con2 = cm.getConnection();
24  Statement stmt2 = cm.createStatement(con2);
    // ... ...
27  ResultSet rs2 = stmt2.ExecuteQuery(balanceQuery);
28  ResultSet maxRs2 = stmt2.executeQuery(maxQry);
29  if (maxRs2.next())
    // ... ...
31  ResultSet minRs2 = stmt2.executeQuery(minQry);
    // ... ...
40  while(rs2.next())
    // ... ...
```

Figure 5.13: A buggy JDBC manipulation example

to a local or a `NotAliased` field" and "a `MaybeAliased` parameter can only be assigned to a local or a `MaybeAliased` field", Fugue can track the aliasing information across method call.

Figure 5.14 [6] shows us an example about how Fugue specifies the API `Socket`. For every object with type `Socket`, it has four possible typestates: `"Raw"`, `"Bound"`, `"Connected"` or `"Closed"`. For every method, Fugue specifies the pre and post condition. For example, the precondition for method `connect()` is that the receiver of the method must in typestate `"Bound"` and after the execution of that method, the typestate of the receiver changes to `"Connected"`. Fugue also specifies the aliasing information for each method and variable concerning if aliasing is allowed or not. For example, for the receiver of the method `open` and `close`, aliasing is prohibited, but for method `send` and `receive`, aliasing is allowed.

Upon checking, Fugue inspects all methods in turn. For every method, Fugue does the data flow analysis over the method control flow graph. At the method entry, the initial typestates representing the pre-condition of the method is deployed. A heap model and corresponding operational semantics on every instructions are defined and is used by the Fugue to check how the method evolves. Among those instructions, method call is checked differently in the sense that Fugue only inspect the callee's declaration and not its body. At the methods exit, Fugue checks if the current typestates agree with the method post-condition. So far, Fugue cannot handle global variables, concurrency features and exceptions.

Bierhoff et al. [9] improves Fugue on several ways. First, by extending the typestate

[6] Since Fugue is based on CLR language, so this specification is different from the FSM specification in Figure 5.11 which is based on Java .

```
[ TypeStates("Raw", "Bound", "Connected", "Closed") ]
class Socket {

 [ Post("Raw"), NotAliased ]
 Socket ();

 [ Pre("Raw"), Post("Bound"), NotAliased ]
 void Bind( string endpoint );

 [ Pre("Bound"), Post("Connected"), NotAliased ]
 void Connect ();

 [ Pre("Connected") ]
 void Send( string data );

 [ Pre("Connected") ]
 string Receive ();

 [ Pre("Connected"), Post("Closed"), NotAliased ]
 void Close ();
}
```

Figure 5.14: A Fugue specification for API `Socket`

refinement ability to the subclasses, they can now support more expressive specifications. Second, by applying the "access permission" annotation method which allows to change state even when aliasing exists (still has some constraints though), they are more flexible than Fugue's `NotAliased/MaybeAliased` annotation method.

Ramalingam et al. [70] use an alternative way to handle API conformance. They specify the API conformance specification with a Java like language `EASL/P`. The original Java program is translated into TVP (a language based on first order logic to specify the operational semantics) program. By applying a parametric shape analysis on the heap while checking through the program control graphs, the checker can dig out possible conformance violations. So far, this method cannot handle resource leak problem and is hard to scale.

Compared to these related works, our method uses real Java as the executable specification which can specify things that finite state machine protocol cannot describe. We based our analysis on the implicit program model (sliced program) rather than on the program control flow graph. For aliasing analysis, we are monitoring the real heap, which means we do not need extra annotation information about the aliasing and we can handle aliasing directly. Furthermore, our method can handle global variables, concurrency features and exceptions.

Chapter 6

Access Rights Verification

6.1 Introduction

Access control mechanisms in software systems are used to supervise access rights to system resources like files, memory, and CPU time. Access control is meant to protect software systems from malicious attacks by granting authorized access to resources and preventing unauthorized access. As more distributed software is deployed, access control plays a bigger role in system security. However, proper use of access control mechanism is far from straightforward. Problems like insufficient access checking or excessive access rights granting are very common mistakes which make software vulnerable to abuse.

Software model checking works on control-intensive verification problems in which the state space is small, but the paths through the space are intricate. Since the verification of access granting is control intensive it is natural to ask if there is a feasible model-checking based verification framework for access rights verification in Java programs.

In order to fulfill the task, that verification framework need s to have the following characteristics:

- It allows examination of all possible execution paths including control flows resulting from raised exceptions.
- It provides abstraction mechanisms that make model checking of large system feasible.
- The back-end model checking engine can handle real programs with language-based access control mechanisms.
- The back-end model checking engine has the power to inspect the Java execution stack [79].

As we can see, our extended Fex can fulfill the first three characteristics. It is possible to customize the Fex to solve this problem. This chapter describes our effort to extend the Fex tool for verifying that access checks are properly used in Java applications.

This chapter is organized as follows: In the next section, we describe the access control mechanism of J2SE version 1.4. In Sections 3, we discuss how we apply the extend Fex for the access rights verification. A case study is given in Section 4. Section 5 introduces the related work and Section 6 concludes.

6.2 Java Access Control Mechanism

In this section, we briefly present the Java 2 access control mechanism using a banking example adopted from [52] with minor modifications. This example is used later to present our verification framework.

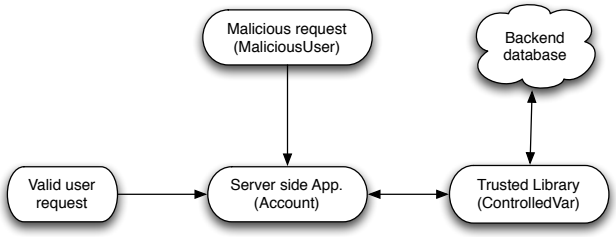

Figure 6.1: On-line banking system: component layout

Figure 6.1 shows the architecture of the banking system. User information including account numbers and balances is stored in a back-end database. Class `ControlledVar`, Figure 6.2, supplies trusted library code to manage the database. This class provides basic operations of `read` and `write`, each protected by a corresponding permission check.

In this diagram, edges represent method invocations, not network requests. The code for the system is assembled from various components, each component comes from a source identified by an URL. The code could be late-bound, perhaps during execution. What matters is that every method call originates from some URL-named codebase. Permissions are then associated with codebases as defined by the developer in a policy file, see Section 6.2.1.

Security sensitive methods call permission checks. Whether a check is successful during execution, depends on a policy file and active method calls on the stack, see Section 6.2.2.

Figure 6.9 presents a server based on the trusted code of Figure 6.2. The server handles client's requests and updates the back-end database. These requests include `getBalance`, `debit` and `credit`.

When a client makes a request to the server, it results in client's code executing in the context of the server to handle the request. Figure 6.3 illustrates sample client's code. If the client code is granted the required permissions then its execution succeeds. Otherwise, an access permission exception will be raised.

```
01 public class ControlledVar {
02    // ... ...
03    ControlledVar (float init, String id) { /* ... ... */ }
04    void write(float f) {
05        WritePermission wp = new WritePermission("write");
06        AccessController.checkPermission(wp);
07                var = f;
08                // do the actual IO work
09    }
10    float read() {
11        ReadPermission rp = new ReadPermission("read");
12        AccessController.checkPermission(rp);
13        // do the actual IO work
14        return var;
15    }
16    void clean () {
17        ReadPermission rp = new ReadPermission("read");
18        AccessController.checkPermission(rp);
19        WritePermission wp = new WritePermission("write");
20        AccessController.checkPermission(wp);
21        this.read();
22        // do the rest actual cleanup work ...
23     }
24 }
```

Figure 6.2: Class `ControlledVar`

```
01 public class ValidUser {
02    public static void main () {
03        Account account = new Account(1000, "001");
04        account.getBalance();
05   }
06 }
```

Figure 6.3: Valid user Code Example

6.2.1 Defining and granting Permissions

In Java 2, all access rights are represented by permission classes. All permission classes are derived from the abstract class `java.security.Permission`. These classes represent access rights to certain system resources. The permission definition usually takes two parameters: *target* and *action*. For example, the following `new` creates `FilePermission` with action `write` to file named `"/home/a.txt"`.

```
FilePermission perm = new java.io.FilePermission("/home/a.txt", "write");
```

Another way to define permission classes is through extending subclasses of class `Permission` such as `java.security.BasicPermission`. For example, the following permission class `WritePermission`

```
01 public class WritePermission extends BasicPermission {
02    public WritePermission(String para) {
03        super(para);
04    }
05 }
```

is used in class `ControlledVar` (Figure 6.2) to protect the `write` operation. This class is used in Figure 6.2 line 5, where we are instantiating a `WritePermission` object `wp` to represent

write access permission. In our example, there are several application specific permission classes: `ReadPermission`, `WritePermission`, `BalancePermission`, `CreditPermission` and `DebitPermission`. Their definitions are analogous to class `WritePermission`.

In Java, most permissions are granted statically by using a security policy file. This policy file contains a sequence of policy clauses. Here is an example:

```
grant CodeBase "www.aBank.com"{
     permission WritePermission "write";
     permission ReadPermission "read";
};
```

This policy clause states that any code executed from the codebase located at the URL `www.aBank.com` has `WritePermission` with the parameter `write` and `ReadPermission` with parameter `read`. A policy file usually includes many `grant` clauses.

In our example, any code attempting to call methods `read` and `write` of class `ControlledVar` must have the respective permissions, `ReadPermission` and `WritePermission`. Such permissions are granted only to code arriving from the URL `www.aBank.com`.

Similarly, valid callers of methods from class `Account` must have all required permissions since this class also comes from a trusted library. In order to be executed, the code of the `ValidUser` in Figure 6.3 must have expected permissions: `BalancePermission`, `DebitPermission` and `CreditPermission` which are granted using entries in the policy file, for example, as

```
grant CodeBase "www.aBank.com"{
     permission BalancePermission "getBalance";
     permission CreditPermission "credit";
     permission DebitPermission "debit";
};
```

Note that the policy file can grant different permissions to code originating in different code bases, but for our example to work class `Account` must come also from the code base granted the `read` and `write` permissions. However, a valid user is not expected to come from a code base that must have all the permissions. Our example will work when the code of valid users comes from a code base granted all the permissions to interact with the `Account` class as follows:

```
grant CodeBase "www.ATMServer.com" {
     permission BalancePermission "getBalance";
     permission CreditPermission "credit";
     permission DebitPermission "debit";
};
```

Note that with this architecture, it is possible for a malicious request from an untrusted code base to attempt to get access to critical operations. The policy file and permissions are meant to prevent such an attacker from succeeding. In this situation, malicious code is essentially the same as that of a valid code:

```
01 public class MaliciousUser {
02    public static void main () {
03        Account account = new Account(1000, "002");
04        account.debit(100);
05    }
06 }
```

What makes it malicious is that it originates from a place not mentioned in the policy file and thus has no permissions granted. Our task is to establish whether such an attack can ever succeed.

6.2.2 Checking Permissions with Stack Inspection

Java's standard library provides special class `java.security.AccessController` with a number of security related methods. Before a critical operation, a method `checkPermission` from `AccessController` can be called to check that a given permission is granted, and throw an exception if not. For example, in Figure 6.2, we first instantiate a permission object `wp` at line 5. At line 6, we check if the code currently executing is granted the permission required for continuing with the `write` operation. If not, an `AccessControlException` is generated.

Java's access control mechanism is built around the concept of stack inspection [79]. Stack inspection works as follows: each class is granted a set of *permissions* to execute certain operations (for example, reading a file or writing to a file, etc) according to the policy file. Usually, every method and its entire call chain must have the required permission granted. In this way, the stack inspection may prevent a program from performing an operation on behalf of unauthorized code. However, this is too restrictive in general and Java also provides a *Privileged* mechanism to relax the permission requirements for a region of code. A privileged region is entered when method `doPrivileged` is called and lasts for the time of the call. For example, lines 26 to 33 in Figure 6.9 form a privileged region which is called when `credit` executes. During execution of the code in the privileged region, JVM marks all frames that were placed on the stack before the current method (`credit` in our case) as privileged.

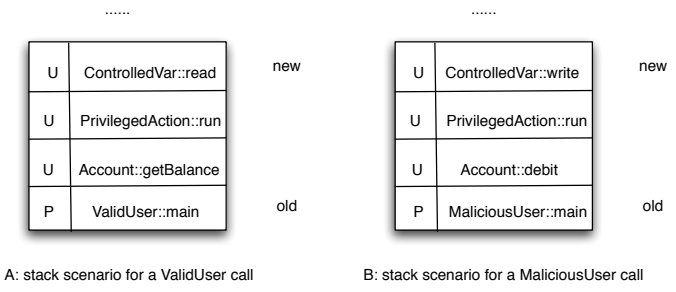

Figure 6.4: Runtime stack demonstration. The essential difference between scenarios A and B is the URL at which the initial method `main` originated.

Method `getBalance` in class `Account` (see Figure 6.9) is acting as the interface to get the credit information—its caller is usually not expected to have `ReadPermission` to directly access the database. A privileged region is deployed here to solve the problem.

Figure 6.4 part A shows a runtime stack snapshot for that scenario. The main entrance of this calling chain is the `main` method from class `ValidUser`, Figure 6.3. This class comes from the URL `www.ATMServer.com` and is granted customer level permissions like `DebitPermission` etc. However, `ValidUser` code has no `ReadPermission` but having the `BalancePermission` granted in the policy file, it calls `getBalance` from class `Account` to do the job.

When the privileged region of `getBalance` is executing, the method `main` that called `getBalance` is marked with a P (for privileged); other frames are marked U (for unprivileged). At the top of the call stack, the method `read` from class `ControlledVar` is about to execute. Because it is protected by a `ReadPermission` check—line 12 in Figure 6.2—the whole stack needs to be inspected; the stack inspection proceeds as follows.

1. Check that class `ControlledVar` has the `ReadPermission`. It has as it comes from `www.aBank.com`.

2. The frame of `PrivilegedAction::run` corresponds to code from the Standard Java API and thus has all permissions.

3. Check that class `Account` has the `ReadPermission`. It has as it comes from `www.aBank.com`.

4. Check that class `ValidUSer` has the `ReadPermission`. Since this frame is marked privileged, JVM skips the check, stack inspection succeeds and the program continues.

6.3 Fex for Access Rights Verification

6.3.1 Access Rights Verification

We call a program *access rights reliable* when there is no possible execution path along which an unauthorized user can eventually get access to critical operations.

Using the Java security model in practice is not straightforward. Since it is up to the code to check if it has permission to do something, Careless security checking often leads to insufficient permission checking and therefore breaches system security. In our malicious code example, operation `write` from class `ControlledVar` directly updates the user's account. Operation `debit` is called by a valid user and to change user's account information by calling `write`. A valid user is granted `DebitPermission` but not `WritePermission` in the policy file. The operation `debit` has to invoke a privileged region as otherwise stack inspection would prevent a valid user from having `write` executed on its behalf by `debit`. However, if—

erroneously—there is no check for `DebitPermission` in `debit` (see Figure 6.9), a malicious user like class `MaliciousUser` can get access to operation `write` by calling `debit`.

The code of `MaliciousUser` originates at an URL which has no permissions yet it can call `Account::debit`. Because of the absence of `DebitPermission` check in `Account::debit` code, the call proceeds to the privileged region. Further stack inspections do not discover any irregularities and finally, the back-end data base is illegally modified, see scenario B in Figure 6.4. Even this small example calls for a technique to verify that a system has appropriate permission checks that ensure security. This is what our framework assists in discovering. How this is achieved is discussed in Section 6.4.

6.3.2 Verification Process

As described in Chapter 4, the verification process for access right verification can be divided into three steps (Figure 4.3):

1. Static analysis and code instrumentation.
2. Program slicing.
3. Model checking.

The purpose of the program instrumentation is to produce a Java program with all possible exception can have a chance to be raise. Program slicing is used to produce a simplified Java program which has significant smaller program state to explore ans still preserves all interesting program properties. Several manually prepared specification Java classes are used to provide the permission granting information and an abstract version of the critical operation. The simplified Java program, along with these executable specification Java classes, are fed into the back-end model checker. Our back-end model checker JPF , which has been customized to have the power to inspect the Java execution stack and perform the stack inspection, can then catch any possible permission violations.

Instrumentation

Malicious code almost always stages an attack by exploiting situations that are unanticipated or rarely encountered. Theses situations are frequently related to the program components handling exceptional situations. Therefore, the ability to examine all possible exceptional control flows is essential in our verification. As described in appendix A.3, after program instrumentation process, the resulting Java program is able to raise all possible exceptions, thus we have the confidence that we are examine the complete program control flow.

Slicing

The instrumented program from step 1 is fed into a slicer to remove all program constructs deemed irrelevant for analyzing access rights. According to chapter 4, The program slicer is now divide all `Java` types into four categories:

1. Control-flow dependent types (CType): e.g. the `Thread` class.
2. Primitive types (*PrimType*).
3. Crucial types (PType): these types which are related to checking access rights permissions, e.g. all subclasses of `java.security.Permission` and the class `java.security.AccessController`.
4. Ignored types (LType) : these types that come from the Java library or from third party application whose source code is not available. Since we do not know how the classes in LType implement the access rights mechanism, we assume that they do it properly. Such classes are trusted and ignored in further verification.

By slicing, the class fields of primary or ignored type are removed. Parameters of primary or ignored type are replaced with a fixed value, e.g. `NULL` for reference types, `2` for integer, etc.

Executable Specification for Access Right Verification

In order to model check the simplified program, the permission granting information (which class is granted with what kind of permission) should be visible to the model checker. In reality, this permission granting information is encoded in a security policy file (see Section 6.2.1), and it is the virtual machine's responsibility to provide correct permission granting information upon the stack inspection process. In our model checking phase, to decode the security policy file needs to take care of too much implementation detail and is therefore impractical. Instead, we use an specification class `ca.ualberta.cs.PermissionRelation` to present the permission granting information to the model checker. This class encodes the permission granting information by using if statements with the name check.

Figure 6.5 is an example for class `PermissionRelation` which encodes the permission granting information for the banking application we introduced in Section 2. In this example, to check if the class `bankexample.Controlledvar` has the permission `ReadPermission`, we only need to call the static method `checkPermissionRelation` from class `PermissionRelation` with the correct parameter `cName` and `pName`. Although this specification class can be translated from the security policy file automatically, we are now doing this manually for convenience.

```
package ca.ualberta.cs;

public class PermissionRelation{
  // ... ...

  public static boolean checkPermissionRelation(String cName, String pName) {

    if (cName.equals("bankexample.ControlledVar") & pName.equals("ReadPermission")) return true;
    if (cName.equals("bankexample.ControlledVar") & pName.equals("WritePermission")) return true;
    if (cName.equals("bankexample.ControlledVar") & pName.equals("BalancePermission")) return true;
    if (cName.equals("bankexample.ControlledVar") & pName.equals("DebitPermission")) return true;
    if (cName.equals("bankexample.ControlledVar") & pName.equals("CreditPermission")) return true;
    // ... ...
    return false;
  }
  // ... ...
}
```

Figure 6.5: Class `PermissionRelation` as the executable specification

Besides the permission granting information, the back-end model checker also need to be aware of which method are considered as the critical operation, the operation which should never be reachable from the outside attacker. In fact, we are not interested in how these critical operation functions, the only thing we are concerned is the reachability of the critical operation. Therefore, we replace every critical operation with its executable specification counterpart whose body contains only one assertion statement:

`assert:contradiction`

This assertion is used to denote that the critical operation is suppose to be unreachable. Figure 6.6 show us an example for the executable specification version of the critical operation for the banking application in Section 2.

```
public class ControlledVar {
   // ... ...
    void write(float f) {
      assert: contradiction
    }

    float read() {
      assert: contradiction
    }
    // ... ...
}
```

Figure 6.6: Executable specification for critical operation

Model Checking

The current version of JPF does not support the stack inspection action. We need to customize the JPF to perform the stack inspection action. how this is achieved is introduced in Section 6.3.3.

Because there might have several entrance points for a certain application, a test harness

`Attacker` which can cover all possible public entrances should also be provided. How to get this test harness is introduced in Section 6.3.4.

In the model checking phase, the simplified program along with the executable specifications which presents the permission granting information and the critical operation are fed into the back-end model checker JPF for examination. JPF used the test harness `Attacker` as the main entrance to explore all possible execution paths, if along any paths, there is a permission violation, a possible access right violation is detected.

6.3.3 Customizing the Model Checker

The sliced program is fed into a model checker, a customized version of Java Pathfinder, JPF [78], to search for access rights violations. Upon detecting a possible access rights violation, JPF dumps out an execution path leading to the offending situation.

JPF is an unusual software model checker in that it directly handles Java byte code. JPF consists of a custom Java virtual machine that executes the byte code and a search engine that guides the execution. Since the states–JVM snapshots—are coded into concise representation, one can use an explicit model checking algorithm to systematically explore all potential execution paths of a program to locate undesired states. In the process, JPF searches for deadlocks, unhandled exceptions and assertion violations. Once a violations is found, JPF reports the entire execution path that leads to the situation.

Java library class `AccessController` depends, among others, on some native code for reading the security policy file. Although the basic JPF cannot handle native code, JPF provides an an extension mechanism called the model Java interface (MJI). MJI is analogous to Java native interface (JNI) and is used to delegate execution from the Java level to the native system level. MJI is used to transfer the execution from JPF controlled VM level into the host VM level. MJI guarantees that once the corresponding method is invoked, the MJI version will be executed instead of the original Java version. This is particularly useful for intercepting calls to native methods or to code which is too complex for model checking. We have implemented a customized MJI version for `java.security.AccessController` called `JPF_java_security_AccessController`.

JPF provides functionality for gathering call stack information at a specific program state. Therefore, we can do the stack inspection in our customized version `checkPermission` procedure in a way presented in Section 6.2.2.

6.3.4 Generating test cases

Our framework automatically generates a test case class named `Attacker` which defines the entry point for JPF. This class forms a test harness providing environmental choices to the model checker by exhaustively calling all publicly accessible program methods. For

example, the generated `Attacker` class example for application `Account` from Figure 6.9 is as in Figure 6.7.

```
public class Attacker {
  public static void main(String[] args) {
    bankexample.Account var;
    var = new bankexample.Account( 2, (java.lang.String)null);
    try { var.getBalance();} catch (Exception e) {}
    try { var.debit(2);}     catch (Exception e) {}
    try { var.credit(2);}    catch (Exception e) {}
  }
}
```

Figure 6.7: Class `Attacker`

The class `Attacker` is generated by using `Java` reflection [35] tools for examining internals of the given application. By using Java reflection, we examine the internal structure of the given Java applications. The generate test harness class `Attacker` instantiates application classes and tries to call all the publicly accessible methods. These methods are considered as the possible program entries for the given application. By doing this, we have the confidence that all possible entry points are checked.

`Attacker` acts as an intruder and by default is granted no permissions. However, the user can grant permissions to this code by modifying the executable specification class `PermissionRelation`. These methods which are considered critical should also be replaced by it's executable specification replacement whose method body is only one statement `assert: false` to identify its unreachable property. In general, the `Attacker` is not allowed to lead to their execution. Every execution path which begins with the main entry of `Attacker` and eventually reaches one of the critical operations forms an access rights violation and is reported.

6.4 Experimental Results

We present some experimental results based on the on-line banking example we introduced in Section 2. Ignoring the `ValidUser` in Figure 6.1, an access permission breech involves three players:

- A trusted library (Class `ControlledVar`, see Figure 6.2), with the functionality to manipulate the back-end database. It has all the permissions.
- A server side application (Class `Account`, see Figure 6.9), with the functionality to perform remote requests. It has all the permissions.
- A malicious agent (Class `Attacker`, see section 6.3.4) which acts as an intruder, trying to get access to critical operations. It has no permissions granted.

We want to verify that there is no possibility of an unauthorized access to critical operations: `read` and `write` from class `ControlledVar`. With the assistance of our verification framework, we find two access rights violations in this program.

The first security flaw is inside the method `debit` from class `Account`. This method contains a privileged region which calls critical operations `read` and `write` from class `ControlledVar`. They are used to directly update user's account information. Since these critical operations are all in the privileged region, operation `debit` can be called to update user's account information. It looks fine since every critical database manipulation operation is protected by corresponding permission checks. However, since there is no permission check for `DebitPermission` in `debit`, a class like `Attacker` which has no permissions can eventually get access to operations `read` and `write` by calling `debit`. This may lead to an illegal database modification.

Our verification framework detects this scenario as an access control flaw as follows. Since class `Attacker` doesn't have any permission granted, any execution which starts at a method from `Attacker` and reaches the critical operation `read` is a violation of the program access rights policy. Every critical operation is replaced by the executable specification whose body contains only one assertion statement: *assert: contradiction* (denoting that the critical operation is suppose to be unreachable.) If JPF is started at the class `Attacker` and the control flow reaches a critical operation, JPF aborts when it reaches the assertion. Upon abort, JPF presents an execution path leading from a method of `Attacker` to a critical operation. The actual call chain is analogous to Figure 6.4 part B.

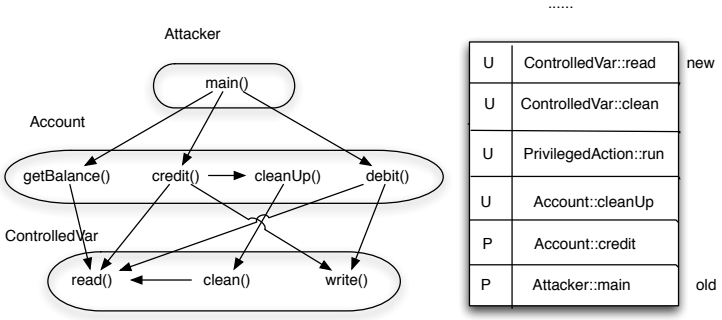

Figure 6.8: Call graph and the runtime stack state for the 2nd access control flaw

The above security flaw is quite trivial in the sense that it can be detected by an experienced programmer playing with a standard Java compiler. On the other hand, our verification framework also identifies another security flaw which other static analysis tools like ESC/Java [34] miss. This security flaw is inside the method `credit` from class `Account`.

By going through the error execution path produced by our model checker, we see the security flaw scenario as follows: the method `credit` is first checking system consistency by calling procedure `checkConsistency`, then it checks if the current user has the `CreditPermission`. If so, the program continues with the actual credit operation, if not, the program throws out an `AccessControlException`. This guarantees that a malicious user which doesn't have proper permissions can't break inside.

There is a subtle error: since the method `checkConsistency` might throw an `IOException`. The careful programmer has placed a `catch` block to perform the clean up by calling `cleanUp`. But this `cleanUp` operation is not protected by any permission checks! Unfortunately, the `cleanUp` method calls `clean` from `ControlledVar`, which in turn calls the critical operation `read`.

The above means that a malicious user eventually gets access to the critical operation `read` by staging an `IOException`. This flaw is detected by our verification framework since the instrumented program triggers all possible `IOException`s. As a result, the backend model checker can go through this implicit control path which begins with the class `Attacker`, and when the control flow eventually reaches the critical operation `read`, the model checker detects a stack violation because the malicious class `Attacker` has no permissions granted. (The JPF running time on this small example is negligible.) Figure 6.8 demonstrates the call graph and the runtime stack snapshot for this situation.

6.5 Related Work

There are several approaches to access rights verification that are closely related to stack analysis. Jensen et al. [52] proposed a framework for verifying general security properties by model checking. They model the program as a *flow graph*. Each node in the flow graph corresponds to a program point. Each edge presents a possible control flow in the program. Since there is no data handled in this model, the operational semantics is defined as a transition system over a control stack consisting of program points. A model checking tool was further developed by Jensen to verify that in all executions all permissions are granted wherever they are being checked.

Following Jensen's pioneering work, Naumovich [66] discusses the same problem using a conservative data flow analysis tool while his program model is basically the same as Jensen's. Koved et al. [54] also propose a technique for computing the access rights requirements by using a precise data flow analysis. Compared to Naumovich's work, they can also model more advanced features, like multi-threading. Bartoletti et al. [6] adopt a control flow graph based static analysis to approximate the access right sets granted at run time. Blanc et al. [8] propose a static analysis tool for detecting access control related security defects. They also provide a formal model on which the correctness of their analysis is proved.

Compared to Jensen's pioneering work and other control flow based static analysis techniques, we offer several advantages.

- Our tool can handle all Java language features including threads.
- We consider all possible exceptions.
- Most of the previous techniques are based on approximate control flow analysis for virtual calls and inter-procedural calls. Our framework is based on model checking actual calls in a simplified program.

6.6 Discussions

We have presented a verification framework for access rights verification that uses a customized version of the JPF model checker for examining call stack and execution paths. First, the investigated program undergoes static analysis and is instrumented such that all exceptions can be easily raised. Then the program is simplified through slicing which retains only control flow and the access rights relevant data. A test harness is automatically generated and the model checker examines all execution paths of the simplified program. A set of manually prepared specification classes states the permission granting information and the critical methods that are not permitted to be accessed by the test suite. If the model checker reaches such a critical method, then an access right violation is reported. With our simplification rules we can still obtain false positives (e.g. due to raising an exception unreachable in the original program).

This framework is directly derived from the extended Fex which is used for verifying event sequence related program properties. Compared to the event sequence program properties such as API conformance verification, access right verification share much in common. Both verification are control-flow determined, differing only in data types relevant for the verified property. For API conformance check, the relevant data could be of any class which is defined as crucial types. For access rights verification, the relevant data is any class derived from the base permission class. We are looking for other control-flow determined applications for this "aggressive" slicing approach.

```
01 public class Account {
02   private ControlledVar  balance; // ... ...
03   public Float getBalance() {
04        BalancePermission bp = new BalancePermission("getBalance");
05        AccessController.checkPermission(bp);
06        AccessController.doPrivileged(
07        /*... Manipulate database by calling Controlled::read ...*/);}
08   public void debit(final float amount) {
09        AccessController.doPrivileged(
10        /*... Manipulate database by calling Controlled::write ...*/);}
11   private void cleanUp() {
12         AccessController.doPrivileged(
13         new PrivilegedAction() {
14         public Object run() {
15         balance.clean();
16         return null;
17         }
18        }
19      );
20    }
21   public void credit(final float amount) {
22   try {
23       checkConsistency(); //might throw IOException;
24       CreditPermission cp = new CreditPermission("credit");
25       AccessController.checkPermission(cp);
26       AccessController.doPrivileged(
27       new PrivilegedAction() {
28       public Object run() {
29          balance.write(balance.read() + amount);
30           return null;
31            }
32          }
33         );
34       }
35   catch(IOException e) {
36      this.cleanUp();
37     }
38    }
39   private void checkConsistency() throws IOException { /* ... ... */ }
40 }
```

Figure 6.9: Class `Account`

Chapter 7

Conclusion and Future Work

7.1 Summary of Result and Contributions

The world is increasingly dependent on software, yet software remains buggy. Many research efforts to improve software reliability have focused on proving the absence of defects. While such software verification has been a goal since the work of Floyd, Hoare and Dijkstra in the 1960s, it is now becoming reality. The shift in focus from total correctness, which is hard to even specify, to common specific properties (e.g., memory safety and deadlock freedom) and new techniques for model checking such as automatic abstraction, decision procedures, program analysis combined with great increases in computing power have resulted in practical verification methods.

Model checking approaches are precise but are choked by state explosion as they track too many facts. Most researchers agree that the best way to attack this problem is to use program abstraction. by applying proper program abstraction, one can have a simplified program (or program model) which not only preserves the interesting program properties but also keeps the program states at a manageable level.

We use the term *event sequence related program properties* to denote these program properties which are related to the execution sequence of the program events. As model checking is an ideal approach for control-flow intensive verification task and event sequence related program properties is indeed control flow intensive, we propose to use a model checking based verification framework for verifying event sequence related program properties. In order to effectively detect subtle errors, this framework must have the following functionality:

- The framework needs to provide the ability to explore all possible control-flow paths including the implicit one such as control-flow raised by an unexpected exceptions.
- The framework needs to conduct the program abstraction techniques to not only preserve the interesting program properties but also keep the program state at a manageable level.

This thesis is focuses on presenting a framework which can achieve the above two goals. This framework combines static analysis with model checking techniques for verifying event based program properties fully automatically. The feasibility and effectiveness of the framework are demonstrated by these case studies. We conclude that this tool is indeed useful for detecting subtle errors.

As we know, handling exceptions is important for software reliability. Up to half of a mature program may be devoted to exception handling. Unfortunately, it is difficult to reason about control-flow paths introduced by exceptions because exceptions do not occur on demand and thus do not lend themselves to traditional verification methods. As event sequence-based program properties are essentially control flow determined, being able to acquire a complete control flow from the program is crucial. To be precise, all exception-triggered program control flows must be included in our verification scope. Our framework deploys a static analyzer for exception instrumentation. By static analysis, we can have all possible exceptions instrumented at the proper places. The back-end model checker can then be aware of these potential exceptions and check the corresponding implicit program control flows raised by these exceptions. Thus we can now have a complete control flow model of the program under investigation.

We use *executable specification* to specify event sequence related program properties. Executable specification is an abstract version of the original implementation. By discarding the irrelevant implementation details and focus on the interesting program properties, executable specification can greatly reduce the program state which need to be explored. As executable specification is usually manually prepared, it can be used to specify all kinds of program properties.

Mindlessly applying program abstraction may either preserve too much details which make the checking process less effective or discards interesting program behaviors which skips subtle errors. We have developed a property guided program abstraction technique which combines data abstraction, loop abstraction and program slicing techniques together to obtain a small yet precise program model. Unlike previous analyses, our program abstraction technique is based on program transformation. The simplified program is guaranteed to have the interesting program properties preserved, providing both precision and scalability. Using this novel technique, we are able to analyze large Java applications with complicated program properties.

These program instrumentation and abstraction techniques are implemented in a tool, Fex, which can verify event sequence related program properties for Java programs. We have used Fex to verify that multi-threaded Java web applications adhere to API conformance rules such as the rule specifying that a socket is eventually closed or that a file be written to only if it was previously opened. Fex is also used to verify the success of access control

checks in Java applications. e.g. there is no possibility that a malicious user can break into an access control protected area.

In short, this work focus on extending the current program instrumentation and abstraction techniques to support software model checking and applying it to verify event sequence related program properties. A summary of the main result and contributions are as follows:

1. The design and implementation of Fex forms a large fraction of this work. We present a novel, property-guided, program abstraction technique which conduct both data abstraction, predicate abstraction, loop abstraction and program slicing to produce a simplified program for checking. We also integrate a program exception instrumentation procedure into our verification process to ensure no implicit control flow is missed.

2. We use executable specifications to describe event sequence related program properties. Previous widely used formalisms like finite state machines can be automatically translated into executable specification and executable specifications can be used for describing more complex program properties such as concurrent modifications and resource leak problems. Executable specifications also provide as an extra program abstraction technique that lets us focus on what we are really interested in.

3. The feasibility and the potential usefulness of the approach are demonstrated by applying Fex to several verification tasks. Both the advantages and limitations of our methodology learned from the experiments are reported. These lessons are important for both the use and future development of the Fex tools.

7.2 Future Work

In this section, we suggest four main directions for future work.

Applying Fex to verify other interesting API conformance rules

Fex has been successfully applied for verifying the conformance of APIs such as `java.net`, `java.io` and `java.sql`. It is necessary to apply Fex to more API conformance case studies to investigate how well Fex works in helping programmers to find subtle conformance errors. For example, Fex can be used to check the conformance rules of `java.util.concurrent.Semaphore` to detect language based concurrent program errors such as deadlocks. Ideally, every java API should have a corresponding executable specification to encode these conformance rules for check.

Fex as a basis for other program verification tasks

Fex conducts the program abstraction by applying an "aggressive" program slicing approach to produce an simplified program which is insensitive to most of data. Therefore

it can only be used to handle control-flow determined program properties such as API conformance check. We are looking for applying Fex to other applications which is control-flow determined. For example, the information flow security problem, in which we need to verify that there is no illicit information flow in programs, can be handled by Fex. As the back-end model checker can systematically exploring all potential execution paths of a program, one can trace the actual information flow over program execution and detecting any illicit information flow.

Improve the error report understanding.

Another research direction which can help the software verification practice is error report understanding. As of now, the error reports generated by software quality tools can often only be understood by tool experts. Ultimately, finding and reporting bugs will not increase software reliability if programmers are unable to understand the reports well enough to fix the bugs. If model checking techniques are to be integrated into software engineering practices, error reports must be improved.

So far the error report generated by Fex is an execution trace which leads to the property violation. The execution trace is actually based on the simplified program. As there is still a big gap between the original program and the simplified program, it need the practitioners substantial efforts to have the error report clearly understand. We think the next step in improving Fex tool would be to building the bridge between the original program and the simplified program so that the trace based error report can actually be shown on the original program instead of on the simplified program. By doing this, we improve the readability of the error report so that the tool can be used by real programmer. We can also adopt a *Counterexample-guided abstraction refinement* methodology which re-examines the counterexample error trace on the original program to either confirm it is a real error or just a false alarm. We believe this method can greatly improve the precision of Fex.

Adding limited data

The current program abstraction strategy that Fex is adopting is rather aggressive in the sense that all primary types such as integers are removed. On the one hand, this give us the benefit to greatly reduce the program state to explore, on the other hand, it also narrows down the application scope of our tool. For example, so far Fex can not be used to verifying arrays with crucial types. The reason is as we have remove all information about integer, we have no idea how the array index works. In other words, all array element are now the same to us.

In order to overcome this limitation, we are thinking about applying a less aggressive program slicer to preserving only limited data. For example, for integer type, instead

of get rid of all integers completely, we now use a small scope such as [-2, 2] to replace the original integer scope. By doing this, we can still make the program state at a manageable level while greatly improves the application scope of the verification tool.

Bibliography

[1] The fizmez web server. see `http://fizmez.com/`.

[2] Kopi java open source compiler. see `http://www.dms.at/kopi`.

[3] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *CAV*, pages 484–487, 2004.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, pages 103–122, 2001.

[6] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Stack inspection and secure program transformations. *Int. J. Inf. Secur.*, 2(3):187–217, 2004.

[7] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 319–331, London, UK, 1998. Springer-Verlag.

[8] Frederic Besson, Tomasz Blanc, Cedric Fournet, and Andrew D. Gordon. From stack inspection to access control: A security analysis for libraries. In *CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations*, page 61, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *OOPSLA*, 2007.

[10] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Form. Methods Syst. Des.*, 25(2-3):167–198, 2004.

[11] Guillaume P. Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Michael R. Lowry, Corina S. Pasareanu, Arnaud Venet, Willem Visser, and Richard Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.

[12] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[13] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Software Eng.*, 26(9):820–836, 2000.

[14] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: an industrial case study. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 431–441, New York, NY, USA, 2002. ACM.

[15] Byeong-Mo Chang. A review on exception analysis and its applications. In *SCI2003*, Orlando, USA, July 2003.

[16] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, New York, NY, USA, 2001. ACM Press.

[17] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257, 1996.

[18] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[19] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[20] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[21] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT press, Cambridge, Massachusetts, 2000.

[22] Michael Colón and Tomás E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 293–304, London, UK, 1998. Springer-Verlag.

[23] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[24] Patrick Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, 2001. Springer-Verlag.

[25] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, January 1999.

[26] Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[27] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.

[28] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP*, pages 465–490, 2004.

[29] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Imperial College, 2000.

[30] Jarno Elonen. The nanohttpd http server. see `elonen.iki.fi/code/nanohttpd`.

[31] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[32] E. Allen Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072, 1990.

[33] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM Press.

[34] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[35] Ira R. Forman, Nate Forman, and Ph. D. Ira R. Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[36] N. E. Fuchs. Specifications are (preferably) executable. *IEE/BCS Software Engineering Journal*, 7(5):323–334, 1992.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[38] Patrice Godefroid. Model checking for programming languages using verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[39] John B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[40] A. M. Gravell and P. Henderson. Executing formal specifications need not be harmful. *IEE/BCS Software Engineering Journal*, 11(2):104–110, 1996.

[41] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[42] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher Order Symbol. Comput.*, 13(4):315–353, 2000.

[43] Klaus Havelund and Willem Visser. Program model checking as a new trend. *STTT*, 4(1):8–20, 2002.

[44] Ian Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Softw. Eng. J.*, 4(6):330–338, 1989.

[45] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[46] G. J. Holzmann and D. Peled. The state of spin. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, number 1102 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[47] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[48] Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Software Eng.*, 28(4):364–377, 2002.

[49] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[50] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems, 2nd Ed*. Cambridge, 2004.

[51] Radu Iosif. Symmetry reduction criteria for software model checking. In *SPIN*, pages 22–41, 2002.

[52] Thomas P. Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[53] J-P. Katoen. *Concepts, Algorithms and Tools for Model Checking*. Arbeitsberichte der Informatik, Friedrich-Alexander-Universitaet Erlangen-Nuernberg, 1999.

[54] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–372, New York, NY, USA, 2002. ACM.

[55] Jet Propulsion Laboratory. Report on the loss of the mars polar lander and deep space 2 missions, 2000.

[56] Leslie Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, 1977.

[57] K. Rustan M. Leino and Wolfram Schulte. Exception safety for c#. In *SEFM*, pages 218–227, 2004.

[58] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[59] Xin Li, H. James Hoover, and Piotr Rudnicki. Towards automatic exception safety verification. In *FM*, pages 396–411, 2006.

[60] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107, New York, NY, USA, 1985. ACM.

[61] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[62] R.A. Maxion and R.T. Olszewski. Improving software robustness with dependability cases. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pages 346–55, 1998.

[63] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[64] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M.D.Ryan, editors, *Modeling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science, pages 3–38. Springer-Verlag, 2001.

[65] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.

[66] Gleb Naumovich. A conservative algorithm for computing the flow of permissions in java programs. *SIGSOFT Softw. Eng. Notes*, 27(4):33–43, 2002.

[67] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.

[68] John Penix, Willem Visser, SeungJoon Park, and et al. Verifying time partitioning in the deos scheduling kernel.

[69] John Penix, Willem Visser, Seungjoon Park, Corina Pasareanu, Eric Engstrom, Aaron Larson, and Nicholas Weininger. Verifying time partitioning in the deos scheduling kernel. *Form. Methods Syst. Des.*, 26(2):103–135, 2005.

[70] G. Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 83–94, New York, NY, USA, 2002. ACM Press.

[71] Steven P. Reiss. Specifying and checking component usage. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 13–22, New York, NY, USA, 2005. ACM Press.

[72] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003.

[73] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[74] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[75] Carl F. Schaefer and Gary N. Bundy. Static analysis of exception handling in ada. *Softw. Pract. Exper.*, 23(10):1157–1174, 1993.

[76] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[77] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[78] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[79] Dan S. Wallach and Edward W. Felten. Understanding Java stack inspection. In *IEEE Symp. on Security and Privacy*, pages 52–63, 1998.

[80] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[81] Eran Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. *SIGPLAN Not.*, 39(6):25–34, 2004.

[82] Kwangkeun Yi. Compile-time Detection of Uncaught Exceptions in Standard ML Programs. In Baudouin Le Charlier, editor, *Static Analysis Symposium*, volume 864, pages 238–254. Springer-Verlag, September 1994.

[83] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Static Analysis Symposium*, pages 98–113, 1997.

Appendix A

Fex Implementation

A.1 Configuration File

The configuration file is used to provide several kinds of information, including:

- What kind of exceptions should be neglected in this verification process?
- What kind of classes/types are considered crucial in this verification process and therefore should be preserved?
- What program do we want to verify?
- Where is the program source code we want to verify?
- Where is the target directory for the generated output?

Fex uses a plain text file as the configuration file. This file contains several lines of rules and for each rule, it must follow the following format:

`<token> value \n`

Where `<token>` is either `<ignore>`, `<crucial>`, `<output>`, `<dir>` or `<import>`. A complete list of so far supported token is in Figure A.1.

The value of token `<ignore>` is used to identify these exceptions which are suppose to be ignored by the static analyzer. By default, our analyzer instruments every possible exception, but the user can choose to skip some kinds of exceptions. The exception labeled with `<ignored>` will not be instrumented by Fex.

The value of token `<crucial>` is used to identify those types which are considered as crucial types (PType, see Chapter 4). For exception reliability check, PType is empty, which means only the control flow related types are preserved. However, for other more complicated verification tasks, those `<crucial>` items are used to inform the slicer which types are considered as inside PType and therefore should be preserved. For example, in API conformance verification, `<crucial>` is used to label these API which we are interested in.

Name	Meaning	Example
<ignore>	to identify the exception which is about to be neglected	<ignore> java.lang.OutOfMemoryError
<crucial>	to identify the class type which is about to be preserved	<crucial> java.net.Socket
<import>	to identify the specific Java package which need to be verified	<import> jnfs.security
<dir>	to identify the path where Java package under verification locates	<dir> /Users/xinli/examples
<output>	to identify the place for output file	<output> /Users/xinli/test

Figure A.1: Example of Configuration Token

The value of token <dir> is used to specify the place where the source code we want to verify lies. The value of token <import> specifies the Java packages which need to be verified. The value of <dir> plus the value of <import> forms a final path. All files with a .java suffix inside the final paths will be processed by Fex. The value of token <output> is used to specify the place for generated Java files.

For example, assume a Java package jnfs.security is the application we want to verify and this package is located under directory /Users/xinli/examples, in order to let Fex find all these Java files which need to be transformed, the value of token <dir> should be set to /Users/xinli/examples and the value of token <import> should be set to jnfs.security. And if the value of token <output> is set to /Users/xinli/test, all generated files will be put under directory /Users/xinli/test/jnfs/security.

Figure A.2 shows an example of a complete configuration file. This file tells us that in this verification process, there are seven types of exceptions need to be neglected. They all labeled with <ignore>. Any statement which might throw these exceptions will not be instrumented. Additionally, we know that the application we want to verify exists in the path /Users/xinli/examples and there are three packages we want to verify. They all labelled by <import> token. Hence all Java files under directory /Users/xinli/examples/jnfs, /Users/xinli/examples/jnfs/security and /Users/xinli/examples/jnfs/security/acl will be processed. The processing result will be put under directory /Users/xinli/test. Finally, since we have two <crucial> items there, we know that objects with class type java.net.Socket or java.io.DataOutputStream are considered as crucial and are preserved.

```
<output> /Users/xinli/test
<ignore> java.lang.OutOfMemoryError
<ignore> java.lang.NegativeArraySizeException
<ignore> java.lang.ArrayIndexOutOfBoundsException
<ignore> java.lang.ArithmeticException
<ignore> java.lang.NullPointerException
<ignore> java.lang.ArrayStoreException
<ignore> java.lang.ClassCastException
<dir>    /Users/xinli/examples
<import> jnfs
<import> jnfs.security
<import> jnfs.security.acl
<crucial> java.net.Socket
<crucial> java.io.DataOutputStream
```

Figure A.2: Example of Configuration File

A.2 AST and Visitor Pattern

The implementation of Fex is based on the Kjc compiler [2], an open sourced Java language compiler and the idea of visitor design pattern. In order to better understand how our program instrumentor and slicer works, we first give a brief introduction to the Kjc compiler, abstract syntax tree and visitor design pattern.

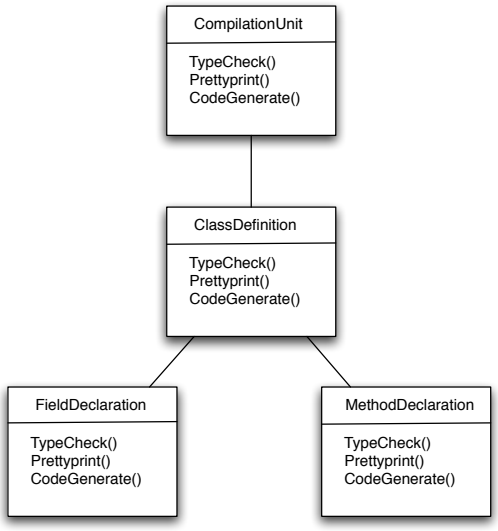

Figure A.3: The AST class hierarchy for Java language in Kjc Compiler (In part)

Kjc compiler parses Java program and encode the parse tree as an abstract syntax tree. An abstract syntax tree (AST) is a directed tree where each inner node represents a meaningful programming language construct and the children of that node represent the sub-components of the construct. It is often used as a compiler or interpreter's internal representation of a program.

In the AST representation, the node that represent the class declaration is different from the node that represent field declaration or method call expression. Hence, there will be one class node for class declaration, another for field declaration, another for method call expression, and so on. For example, Figure A.4 shows part of the AST class node hierarchy tree for Java program. A Java program is composed by class declaration operation. Each class declaration can contain field declaration, method declaration etc.

The compiler needs to perform all all kinds of operations on abstract syntax trees for syntax or semantic analyses such as type checking, pretty printing, code generation and code instrumentation. These operations could be associated with the AST node just as Figure A.4 shows. However, it is confusing to have type-checking code mixed with pretty printing code. A better way to do this would be to present each analysis operation (belonging to a separate compiler feature) separately, leaving the classes in the AST representation independent of the operations that apply to them.

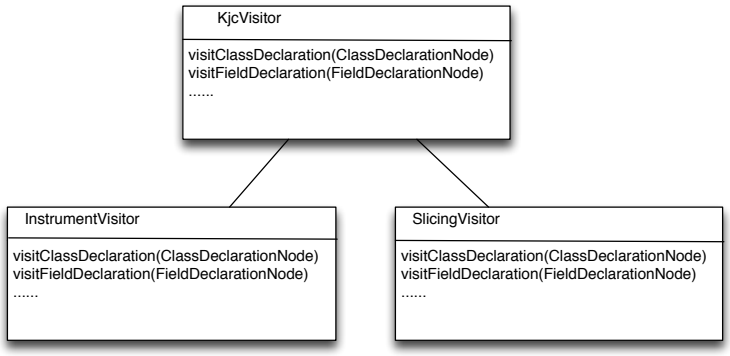

Figure A.4: `KjcVisitor` class hierarchy

The Kjc compiler provides access to the AST through an implementation of the Visitor design pattern [37]. The purpose of the Visitor Pattern is to encapsulate an operation that you want to perform on the elements of a data structure without changing those structures. The word "Visitor" refers to the ability of performing operation without changing anything from the data structure.

The visitor pattern lets you have the separation of operations and structures by packaging related operations from each class in a separate object, called a `Visitor`, Each node in the AST can accept a visitor, which sends a message to the visitor which includes the node's class. The node makes a call to the visitor, passing itself in, and the visitor executes its operation on the node.

```
public interface KjcVisitor {

  void visitClassDeclaration( ... );

  void visitClassBody( ... );

  void visitFieldDeclaration( ... );

  void visitMethodDeclaration( ... );

  void visitKopiMethodDeclaration( ... );

  void visitWhileStatement( ... );

  void visitVariableDeclarationStatement( ... );

  void visitVariableDefinition( ... );

  void visitTryCatchStatement( ... );

  void visitTryFinallyStatement( ... );

  void visitThrowStatement( ... );

  void visitReturnStatement( ... );

  void visitIfStatement( ... );

  void visitExpressionStatement( ... );

  void visitExpressionListStatement( ... );

  void visitMethodCallExpression( ... );

  public void visitBinaryExpression( ... );

   ... ...
}
```

Figure A.5: Code sample for interface KjcVisitor

For example, a compiler that pretty print a program using the visitor pattern would create a `PrettyPrintVisitor` object and call the `accept` operation on the abstract syntax tree with that object as an argument. Every nodes on the AST hierarchy would implement an `accept` for calling back on any `Visitor` instance. For instance, a class declaration node calls `visitClassDeclaration` operation on the `Visitor`, while a method call expression calls `visitMthodCallExpression`. Consequently, the `PrettyPrint` operation that used to be in class `classDeclarationNode` is now the `visitClassDeclaration` operation on `PrettyPrintVisitor`.

Therefore, the visitor pattern results in two interdependent class hierarchies: one for the elements being operated on (AST class node hierarchy in our case) which is literary untouched. And one for the visitors that define operations on the elements (the `PrettyPrintVisitor` hierarchy in our case) which can be quickly expended for a new operation.

Because Kjc compiler works on the AST for more than just one operation, an interface class `KjcVisitor` is presented for all visitors (See Figure A.5). This interface declares an abstract operation based on the AST hierarchy. New functionality can be added simply by defining new `KjcVisitor` subclasses. For example, the class `InstrumentVisitor`, which is used to perform the program instrumentation, and the class `SlicingVisitor`, which is used to perform program slicing, are defined as subclasses of interface `KjcVisitor`.

A.3 Program Instrumentation

The purpose of program instrumentation in Fex is to make sure that every possible exceptions can be raised. In general, it is difficult to gather all exception related information because exceptions do not occur on demand. Static analysis is an efficient way to cure this problem. By approximate the program behavior at the run-time, a static analyzer can gather the exception related information such as what kind of exception that might be raised at which program points. Our exception analysis is based on the Jex tool [73], a static analysis tool for analyzing exception flow in Java programs.

Jex is a static analysis tool that provides information about the exceptions that can be raised in a Java program. For each method of each class, Jex outputs a description of all exceptions that can be raised in the method. The description shows the origin of each exception. Jex is built on top of Kjc, the Kopi open source Java compiler. In fact, Jex is implemented as an extension to the Kjc project. Jex takes as input a set of Java source files and a configuration file and produces, for each Java class, a human-readable Jex file. A Jex file contains, for each method in the associated Java class, a view of the exception flow. Figure A.6 is a Jex file example for a class constructor.

From this example, we see that an `SecurityException` can be raised as a result of the call to method `checkWrite` of class `SecurityManager`. Furthermore, in the `try` block, `IOException` can result from the calls to methods `openAppend` and `open` of class `FileOutputStream`, and a `NullPointerException` can be raised by the run-time environment. The `catch` clause declares to catch `IOException` and in the corresponding block, a `FileNotFoundException` is explicitly raised using the keyword `throw`. The view of exception flow produced by Jex is more precise and more complete than the information available through exception interfaces i.e, the "throws" list in method declarations. Our static analyzer tool can reuse this exception information for our verification purpose.

```
 <init>(java.lang.String,boolean) throws IOException
{
   java.lang.SecurityException:SecurityManager.checkWrite(java.lang.String);
   try
   {
      java.io.IOException:java.io.FileOutputStream.openAppend(java.lang.String);
      java.io.IOException:java.io.FileOutputStream.open(java.lang.String);
      java.lang.NullPointerException:*environment*;
   }
   catch( java.lang.IOException )
   {
      throws java.io.FileNotFoundException;
   }
}
```

Figure A.6: A Jex file example for a class constructor

Based on Jex, our tool is organized around three central functions: parsing and type checking Java files, extracting exceptions, and instrumenting exceptions at the proper places. The parsing and type checking of Java source files is performed by the Kjc compiler. This compiler operates like the standard Sun Java compiler, taking as input a series of Java source files. The Kjc compiler parses the Java files and produces an abstract syntax tree (AST) for each file. Based on the generated AST, The Kjc compiler typechecks the code, which evaluates the type of all expressions in the AST. We then implement the extraction of exception information and the exception instrumentation as one instances of `KjcVisitor`, namely `InstrumentVisitor`.

For environment-generated exception types, we extract the corresponding exception based on the Java language specifications [4] and instrument the possible exception right after the statement which might raise that exception. For example, a statement a/b which attempts a division operation might raise `ArithmeticException` and therefore is instrumented as:

```
a/b;
if (Verify.getBoolean()) throw new ArithmeticException();
```

Thus, not matter what the real value of a and b are, we always considered the possibility of divide by zero error and therefore raises `ArithmeticException`. To do this we need to implement the `visitBinaryExpression()` from class `InstrumentVisitor` to ensure every time when conducting division operation, an `ArithmeticException` exception is instrumented.

For each method call, we have to determine if there is any possible exceptions propagated from the method. The analyzer first determines if the method source code is available. If available, the analyzer instruments the method directly; the model checker can take care of the exception handling and propagating at run time. If not, the analyzer instruments the exception interface extracted from the byte code of that method to guarantee that the exception has a chance to be raised. For example, a method call `fis.read`, where object `fis` belongs to class `java.io.FileOutputStream`, might raise `IOException`. We need to

instrument this statement as:

```
if (Verify.getBoolean()) fis.read();
else throw new IOException();
```

This instrumentation style can precisely mimic the behavior of method `fis.read()`, which has two exclusive consequences.

- The method terminated successfully.
- The method terminated abnormally and an exception is raised.

However, for the operation which might raise more than one exceptions, we need to adopt another instrumentation style. For example, a method call `socket.bind()`, where object `socket` belongs to class `java.net.Socket`, might raise both `IllegalArgumentException` and `IOException`. We then need to instrument this statement as:

```
if (Verify.getBoolean()) socket.bind();
else if (Verify.getBoolean()) throw new IOException();
     else throw new IllegalArgumentException();
```

In this way, we can precisely mimic the behavior of method `socket.bind()`. We do not need inter-procedural analysis to determine the possible exception propagation since this is effectively achieved in the model checking phase. Our analysis only need to traverse the AST twice.

For the method call which fulfills the resource release operation, we are adopting an alternative way for the exception instrumentation. Technically, every resource release method may fail, causes runtime exception. That means every application has a tiny chance to leak resource. As we are only verifying that the programmer did not forget to release resource, whether the release operation success or not is out of the verification scope. Therefore, we instrument every resource release method in a way that the method is guaranteed to be successful. For example, instead of instrument `FileInputStream` method `close` in a traditional way as

```
if (Verify.getBoolean()) is.close();
else throw new java.io.IOException();
```

We instrument it as

```
is.close();
if (Verify.getBoolean()) throw new java.io.IOException();
```

These instrumentation process is fulfilled by coding method `visitMethodCallExpression` in class `InstrumentVisitor`.

Jex is designed to address all exceptions that can be raised in a Java program, so it supports both checked and unchecked exceptions. By default, we consider all possible exceptions flagged by Jex (although not all possible exceptions are reported by the tool) but we also can specify which exceptions to ignore. For example, the unchecked exception, `OutOfMemoryError` could be raised after every `new` operation and method call. The user might want to turn off the instrumentation of this exception until the more common modes of failure have been addressed.

A.4 Program Slicing and Abstraction

Figure A.7 presents the abstract syntax of core Java following the definition in [29]. We would like to perform a source to source program transformation to get a simpler program with less program states to explore. The simplified program needs to follow the program model whose abstract syntax is defined in Figure A.8.

Prog	::=	*Class**
Class	::=	**class** CId **extends** CName
		{*CMember**}
CMember	::=	*Field* | *Method*
Field	::=	*VarType* VarId;
Method	::=	*MHeader MBody*
MHeader	::=	(**void** | *VarType*) MId ((*VarType PId*)*) **throws** CName*
MBody	::=	{*Stmts* [**return** *Expr*]}
Stmts	::=	(*stmt*;)*
Stmt	::=	**if** *Expr Stmts* **else** *Stmts*
	|	*Var* = *Expr* | *Expr*.MName(*Expr**)
	|	**throw** *Expr*
	|	**while** *Expr Stmts*
	|	**try** *Stmts* (**catch** CName Id *Stmts*)* **finally** *Stmts*
	|	**try** *Stmts* (**catch** CName Id *Stmts*)$^+$
Expr	::=	*Value* | Var | *Expr.MName(Expr***)*
	|	**new** CName() | **this**
Var	::=	Name | *Expr. VarName* | *Expr[Expr]*
Value	::=	*PrimValue* | *RefValue*
RefValue	::=	null | ...
PrimValue	::=	*intValue* | *charValue* | *boolValue* | ...
VarType	::=	*PrimType* | CName
PrimType	::=	**bool** | **char** | **int** | ...

Figure A.7: Abstract Syntax of Core Java

Compared to the concrete Java program model in Figure A.7, the syntax of the simplified Java model has been changed in several places:

1. Variable types are now divided into four categories:

   (a) Control-flow dependent types (CType). Includes program defined classes (CId from syntax definition) and all subclasses from class `Throwable` and class `Thread`.

(b) Primary types.

(c) Crucial types (PType).

(d) Ignored types (LType). All other types besides (a), (b) and (c).

And the ignored types (LType) has been removed from the simplified program model.

2. In the concrete model, he *PrimValue* is divided into *intValue*, *charValue*, *boolValue* etc. In the simplified program model, we stuff the predefined value for each primary value. For instances, we stuff 2 for *intValue*, `'a'` for *charValue*, etc. Therefore, the *PrimValue* in the simplified model is fixed.

3. The loop construct in the concrete model is now replaced in the simplified program model by a fixed iteration loop that executes each loop only a fixed number of (in our case, 2) times.

Prog	::=	*Class**
Class	::=	**class** CId **extends** CName
		{*CMember**}
CMember	::=	*Field* | *Method*
Field	::=	*VarType* VarId;
Method	::=	*MHeader MBody*
MHeader	::=	(**void** | *VarType*) MId ((*VarType PId*)*) **throws** CName*
MBody	::=	{*Stmts* [**return** *Expr*]}
Stmts	::=	(*stmt*;)*
Stmt	::=	**if** *Expr Stmts* **else** *Stmts*
	|	*Var* = *Expr* | *Expr*.MName(*Expr**)
	|	**throw** *Expr*
	|	**for (int Findex = 0; Findex** $<$ **2; Findex++)** *Stmts*
	|	**try** *Stmts* (**catch** CName Id *Stmts*)* **finally** *Stmts*
	|	**try** *Stmts* (**catch** CName Id *Stmts*)$^+$
Expr	::=	*Value* | Var | *Expr.MName(Expr***)*
	|	**new** CName() | **this**
Var	::=	Name | *Expr. VarName* | *Expr[Expr]*
Value	::=	*PrimValue* | *RefValue*
RefValue	::=	null | ...
PrimValue	::=	`2` | `'a'` ...
VarType	::=	*PrimType* | CName (iff CName $\in$ CType $\cup$ PType)
PrimType	::=	**bool** | **char** | **int** | ...

Figure A.8: Abstract Syntax of Simplified Java for Verifying Exception Reliability

Besides the program models, we also need to define the transformation rules for the program to describe how we transform a Java program from its concrete model to the simplified model. Mechanically, our program slicer works as follows: First, the Java file under investigation is parsed and the corresponding abstract syntax tree (AST) is generated. The program slicer then traverse the AST, for every inner node which satisfied the precondition

of a transformation rules, that rule is executed and the corresponding program constructs is rewrote. When the traverse terminates, we get a simplified program.

The definition of these program transformation rules are based on the abstract syntax from Figure A.7. A rule of the form

$$[\ \mathrm{A}\ ]\ \mathrm{B}\quad \frac{\mathrm{C}}{\mathrm{D}}$$

means that when visiting program construct A, and while condition B is true, program fragment C (one of the variants of A) is be transformed to D. In these rules, we frequently use the following two operations

- `type(`*expr*`)` returns the type of parameter *expr*.
- `eval(`*expr*`)` returns true iff *expr* is not CType or Ptype related.

In conditional statements, if the guard expression does not involve control flow type or crucial type manipulation (that is `eval(`*expr*`)` returns true), we use `Verify.getBoolean()` to replace the *expr* to ensure that model checking examines all possible paths.

The slicing rules are used to transform a program construct from the original model (Figure A.7) to the simplified model (Figure A.8). Here, we follow the type division category from Chapter 4. For exception reliability check which does not contain any crucial type (PType), one can simply replace the PType with empty set. Here we list all 15 slicing rules. These rules are encoded in class `SlicingVisitor`, which is a subclass of the interface `KjcVisitor`. While traverse the AST of a specific `Java` program, `SlicingVisitor` slice the original program and produce the simplified program.

$$[\mathit{Field}]\ \ \mathit{VarType} \notin \mathrm{PType}\quad \frac{\mathit{VarType}\ \ \mathrm{VarId}}{[\ ]} \tag{A.1}$$

$$[\mathit{Field}]\ \ \mathit{VarType} \in \mathrm{PType}\quad \frac{\mathit{VarType}\ \ \mathrm{VarId}}{\mathit{VarType}\ \ \mathrm{VarId}} \tag{A.2}$$

The above two rules concerns class fields: every class field that is not a PType is removed. However, the class field that is a PType is preserved.

In the actual implementation, the above three rules concerning class field declaration is fulfilled by implementing method `visitFieldDeclaration` from class `SlicingVisitor`.

$$[\mathit{MBody}]\ \ \texttt{type}(\mathit{Expr}) \in \mathrm{LType}\quad \frac{\{\mathit{Stmts}\ [\mathbf{return}\ \mathit{Expr}]\}}{\{\mathit{Stmts}\ [\mathbf{return}\ \texttt{null}]\}} \tag{A.3}$$

This rule concerns return statements: if the type of *Expr* belongs to the ignored type (LType), we return `null` instead.

$$[\mathit{MBody}]\ \ \texttt{type}(\mathit{Expr}) \in \mathit{PrimType}\quad \frac{\{\mathit{Stmts}\ [\mathbf{return}\ \mathit{Expr}]\}}{\{\mathit{Stmts}\ [\mathbf{return}\ \mathit{PrimValue}]\}} \tag{A.4}$$

When the type of the return expression is primitive, then a predefined value of the corresponding primitive type (see below) is returned.

$$[MBody]\ \texttt{type}(Expr) \in CType \cup \text{PType} \quad \frac{\{Stmts\ [\textbf{return}\ Expr]\}}{\{Stmts\ [\textbf{return}\ Expr]\}} \tag{A.5}$$

When the type of the return expression belongs to the control flow dependent type (CType) or crucial type (PType), the whole statement is preserved as untouched.

In the actual implementation, the above three rules concerning return statement is fulfilled by implementing method `visitReturnStatement` from class `SlicingVisitor`.

$$[Stmt]\ \texttt{eval}(Expr) \quad \frac{\textbf{if}\ \ Expr\ \cdots}{\textbf{if}\ \ \texttt{Verify.getBoolean()}\ \ \cdots} \tag{A.6}$$

If the guard expression *Expr* is not CType relevant, it is replaced by `Verify.getBoolean()`.

$$[Stmt]\ \neg\texttt{eval}(Expr) \quad \frac{\textbf{if}\ \ Expr\ \cdots}{\textbf{if}\ \ Expr\ \cdots} \tag{A.7}$$

If the guard expression *Expr* is CType or relevant, it is kept as untouched.

In the actual implementation, the above two rules concerning if statement is fulfilled by implementing method `visitIfStatement` from class `SlicingVisitor`.

$$[Stmt]\ \texttt{eval}(Expr) \quad \frac{\textbf{while}\ \ Expr\ \ Stmts}{\textbf{for (int Findex = 0; Findex < 2; Findex++)}\ Stmts} \tag{A.8}$$

For while loops, if the guard expression *Expr* is not CType relevant, it is replaced by a fixed iteration loop. Therefore, the loop body is executed fixed times. We choose the magic number 2 here because, if we choose 1, the loop construct is actually downgrade to an if statements which is not good for unveil the loop related errors. For example, one may close a resource in a iteration and try to use it in another iteration. There is no way to find this kind of conformance violation by traverse the loop once once. And since the program properties we want to verify is loop index insensitive, choosing the number greater than 2 makes redundant iteration.

$$[Stmt]\ \neg\texttt{eval}(Expr) \quad \frac{\textbf{while}\ \ Expr\ \ Stmts}{\textbf{while}\ \ Expr\ \ Stmts} \tag{A.9}$$

For while loops, if the guard expression *Expr* is CType or PType relevant, it is kept as untouched.

In the actual implementation, the above two rules concerning while statement is fulfilled by implementing method `visitWhileStatement` from class `SlicingVisitor`.

$$[PrimValue] \quad \frac{intValue\ |textitboolValue| \cdots}{\texttt{2}\ \mid\ \texttt{true}\ \mid \cdots} \tag{A.10}$$

Every primitive value is replaced by a predefined value, if it is going to be preserved. For instance, we use 2 to replace any **int** value.

$$[Stmt]\ \texttt{type}(Expr) \in PrimType \cup \text{LType} \quad \frac{Var = Expr}{[\,]} \tag{A.11}$$

All assignment statements over primitive types or library type variables/objects are removed.

In the actual implementation, the above rule concerning primitive type is fulfilled by implementing several methods from class `SlicingVisitor` which might manipulate the program construct with primitive types such as method `visitArguments`.

$$[Stmt]\ \texttt{type}(Expr) \in CType \cup \text{PType} \quad \frac{Var = Expr}{Var = Expr} \tag{A.12}$$

All assignment statements over control flow related types (CType) or crucial types (PType) objects are kept as untouched.

In the actual implementation, the above rule concerning assignment statement is fulfilled by implementing method `visitAssignmentExpression` from class `SlicingVisitor`.

$$[Stmt]\ \texttt{type}(Expr) \in \text{LType} \quad \frac{Expr.MName(Expr^*)}{[\,]} \tag{A.13}$$

Each irrelevant call is removed (as its exceptions have been already instrumented).

$$[Stmt]\ \texttt{type}(Expr_1) \notin \text{LType}\ \ \&\ \texttt{type}(expr) \notin PrimType \cup \text{PType}$$
$$\frac{Expr_1.\text{MName}(expr,\ Expr^*)}{Expr_1.\text{MName}(\texttt{null}, Expr^*)} \tag{A.14}$$

If we invoke a method not from a library, and the parameter is not of primitive type or PType, we use `null` to replace that parameter.

$$[Stmt]\ \texttt{type}(Expr_1) \notin \text{LType}\ \ \&\ \texttt{type}(expr) \in \text{CType} \cup \text{PType}$$
$$\frac{Expr_1.\text{MName}(expr,\ Expr^*)}{Expr_1.\text{MName}(expr,\ Expr^*)} \tag{A.15}$$

If we invoke a method not from a library, and the parameter is from CType or PType, the parameter is kept as untouched.

In the actual implementation, the above three rules concerning method call expressions is fulfilled by implementing method `visitMethodCallExpression` from class `SlicingVisitor`.

In addition to the above, other `Java` program constructs not included in Figure A.8 are handled as follows:

1. Abstract classes and interfaces are sliced like a standard class.

2. Array types are sliced in the same way as the basic array type.

3. All thread related program constructs are preserved since they are essential for program execution. This can be seen as treating them like CType in their slicing rules.

Since we relax the guards of all program control statements, there is a possibility that we generate false alarms; that is, we generate an assertion violation execution path which never happens in the original program. It requires substantial human effort to inspect the output of the model checker in order to dismiss a false alarm. Reducing the frequency of false alarms is one of our future goals.

A.5 Model Checking

We use Java Pathfinder (JPF) as our back-end model checking engine. By default, JPF checks for unhandle exceptions. Hence when feeding with the simplified programs, it can be used directly for the exception reliability verification. For the event sequence related program properties, these event sequence constraints are encoded inside the executable specification classes. JPF takes the simplified program as well as the executable specification classes to exhaustively check if there is any assertion violations (event sequence constraints are encoded as assertion checks inside the executable specification). If any violation is detected, JPF dumps out the execution trace which lead to the violation.

In general, JPF works on the Java byte code and reads all necessary Java class files from the default place. However, in event sequence related program property verification, we are suppose to use the executable specification classes, which is an abstract implementation of the original classes to replace all these concrete classes. For example, in order to perform the conformance check for API `java.io.FileInputStream`, the concrete implementation of that API should be replaced by the executable specification class as in Figure 4.2.

In order to let JPF use these executable specifications instead of the concrete classes, these executable specification should be placed at a proper place. If the executable specification comes with the user defined class, it can be directly feed into the JPF . However, if the executable specification is meant to replace the standard Java API which the JPF has its own copies, these copies should be replaced. JPF keeps its own copies of standard Java API [1] under the directory `\JPFroot\env\jpf` [2], by replace the corresponding JPF copy with the executable specification, JPF can recognize the new class and model check it. For example, to verify the conformance rules for `java.io.FileInputStream`, the corresponding executable specification should be placed under directory `\JPFroot\env\jpf\java\io`.

Under some circumstance, we also need to customize the JPF to fit our requirement.

- The first customization is adding the support for `finalize` method. the standard JPF does not support the `finalize` method due to the performance consideration.

[1] So far, JPF only supports limited standard Java API.
[2] Here, `JPFroot` refers to the directory where JPF is installed.

However, in API conformance verification, the `finalize` method is used to regulate the proper place for checking if the resources are correctly released. Hence, we have to modify the model checker to support `finalize` i.e. to guarantee that every time when an instance from a class which defines `finalize` method dies, that `finalize` will be executed. We found out as long as we keep the implementation of that `finalize` as concise as possible (so far, our implementation of the `finalize` method only involves assertion check), the performance penalty was negligible.

- The second customization is add the support for dynamic stack inspection. The permission check operation in Java is fulfilled by the Java library class `AccessController`. However, the concrete implementation of `AccessController` depends on some native code for reading the security policy file. hence the standard JPF can not perform the stack inspection.

  We use one of JPF's extension mechanism called model Java interface (MJI) to solve this problem. MJI is analogous to Java native interface (JNI) and is used to transfer the execution from JPF controlled VM level into the host VM level. MJI guarantees that once the corresponding method is invoked, the MJI version will be executed instead of the original Java version. This is particularly useful for intercepting calls to native methods or to code which is too complex for model checking. We have implemented a customized MJI version for `java.security.AccessController` called `JPF_java_security_AccessController` to fulfill the real permission check task. The permission relation is acquired from the executable specification class `ca.ualberta.cs.PermissionRelation`.

Appendix B

Redundant Exception Handler Check

Usually every handler in a program is intended to catch some particular exceptions, and if a handler does not catch any, then it is likely that we have a redundant handler.

```
01 import java.io.*;
02 public class Example2 {
03    public static void main(String[] args) throws IOException {
04        FileInputStream fis = new FileInputStream(args[0]);
05        try{
06            int i = fis.read();
07            while(i != -1) {
08                System.out.print((char)i);
09                i = fis.read();
10            }
11            fis.close();
12        }
13        catch (FileNotFoundException ex) {
14            System.err.println(ex);
15        }                                                                    2
16        catch (IOException ex) {
17            System.err.println(ex);
18        }
19    }
20 }
```

Figure B.1: Example of Redundant Exception Handler

Figure B.1 shows us an example of a program with unreachable exception handler. The catch handler from line 13 to line 15 is redundant because from the corresponding try block (line 05–12), a `FileNotFoundException` would never be raised. So this program is not exception handling reliable. This kind of redundant exception handler flaw usually can not be detected by a standard Java compiler[1], but an exception analysis tool such as Jex or our framework is able to do so.

For exception reliability problem, our back-end model checker can detect any uncaught exceptions automatically. but it cannot detect redundant exception handlers directly. This

[1]Some redundant exception handlers, such as a handler for a subclass right after the handler for a super class, are actually dead code and therefore can be detected by a standard Java compiler.

is because the model checker can only sense all program constructs that are reachable. However, the redundant exception handlers are the program constructs that are not reachable. An extra work is needed here.

```
Unreachable: Exception handler for java.io.FileNotFoundException
             at Example2.java: line 18

==================================
  1 Error Found: unreachable exception handler
==================================
```

Figure B.2: Error report for redundant handler

We solve redundant exception handler in the following way: at static analysis session, the analyzer can label all exception handlers and form a set `AllHandler`. At a later stage, the model checker can label all reachable exception handlers and form a set `ReachableHandler`. The difference of these two sets `AllHandler - ReachableHandler` are those exception handlers which are not reachable, i.e. redundant exception handlers. We provide an extra filtering tool for detecting redundant exception handlers. It works as follows. At program instrumenting stage, our analyzer generates a text file as a byproduct, which records all exception handlers. For example, for the program from Figure B.1, this text file looks like:

```
Handler 1 (java.io.FileNotFoundException) at Example2.java: Line 8
Handler 2 (java.io.FileNotFoundException) at Example2.java: Line 18
Handler 3 (java.io.IOException) at Example2.java: Line 22
```

When doing the model checking, We supervise the JPF to record every exception handler it has visited and write it to another text file. For the program from Figure B.1, it looks like:

```
ReachableHandler 1 (java.io.FileNotFoundException) at Example2.java: Line 8
ReachableHandler 2 (java.io.IOException) at Example2.java: Line 22
```

The filtering tool can then compare these two text files and reports those unreachable exception handlers. For this particular case, The output from our framework for this redundant exception handler is as in Figure B.2 [2].

[2]The line numbers in this error report are based on a sliced program, not the original program.