Exception Analysis for Multithreaded Java Programs *

Sukyoung Ryu and Kwangkeun Yi
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1 Kusong-dong Yusong-gu, Daejeon 305-701, Korea
{puppy, kwang}@ropas.kaist.ac.kr

Abstract

This paper presents a static analysis that estimates uncaught exceptions in multithreaded Java programs. In Java, throwing exceptions across threads is deprecated because of the safety problem. Instead of restricting programmers' freedom, we extend Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions in the input programs.

Our analysis consists of two steps. The analysis firstly estimates concurrently evaluated expressions of the multithreads in Java programs by the synchronization relation among the threads. Using this concurrency information, program's exception flow is derived as set-constraints, whose least model is our analysis result. Both of these two steps are proved safe.

1. Introduction

Java [8] offers both exception handling facilities and multithreading mechanisms. Exception facilities allow the programmer to define, throw and catch exceptional conditions. Exceptions are first-class objects in Java. Like normal objects, they can be defined by classes, instantiated, assigned to variables, passed as parameters, etc. Java provides a simple and tightly integrated support for multithreaded programs. A concurrent program consists of multiple threads that run independently at the same time.

However, Java does not provide a good method to use these two features together. The only one way for a thread to throw an asynchronous exception to another thread is invoking `Thread.stop()`. When the `stop` method of a thread c_1 is invoked by another thread c_2, the exception `ThreadDeath` is thrown to the thread c_1 and c_1 is stopped. But `Thread.stop()` is deprecated from JDK1.2 [1] because it is unsafe [2]:

> "Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) ... Unlike other unchecked exceptions, `ThreadDeath` kills threads silently; thus, the user has no warning that his program may be corrupted."

Instead of throwing exceptions across threads, Java recommends a polling method. When a thread kills another thread, the former simply sets some flag to indicate that the latter should stop running. The latter thread should check this flag regularly and stop if the flag is set.

Our Work

In this paper, we extend Java language to support throwing exceptions across threads and present a static analysis that estimates uncaught exceptions in multithreaded Java programs. In contrast to Java's restriction on throwing asynchronous exceptions because of the safety problem, we give Java programmers the freedom of programming and provide them with a tool to check the safety of the programs.

Our analysis consists of two steps: analyzing Java programs' concurrency information and then exception flow. First, the analysis estimates expressions which may be evaluated concurrently in each thread. And then it analyzes the programs' exception flow with the pre-analyzed concurrency information.

Consider the example in Figure 1. When "`main.stop(new Exception())`" is called in the thread `ThreadOne`, the exception `Exception` is thrown to the thread `ThreadMain`. In order to analyze the expressions in `ThreadMain` to which the exception is thrown, our analysis estimates the expressions in `ThreadMain` which may be evaluated concurrently with `main.stop` in `ThreadOne`. Using the `wait` and `notify` relation among `ThreadMain`, `ThreadOne` and `ThreadTwo`, our concurrency analysis estimates

*This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.

```
class ThreadOne extends Thread {
    Object objectO;
    Thread main;
    public ThreadOne (Object objectO,
                      Thread main) {
        this.objectO = objectO;
        this.main = main;
    }
    public void run() {
        try {
            synchronized (objectO)
                { objectO.wait(); }
        } catch (InterruptedException e) {}
            // asynchronous exception throw
        main.stop(new Exception());
    }
}

class ThreadTwo extends Thread {
    Object objectO;
    Object objectP;
    public ThreadTwo (Object objectO,
                      Object objectP) {
        this.objectO = objectO;
        this.objectP = objectP;
    }
    public void run() {
        try {
            this.sleep(1000);
        } catch (InterruptedException e) {}
        synchronized (objectP)
            { objectP.notify(); }
        synchronized (objectO) {
            objectO.notify();
            System.out.println
              ("\nobjectO.notify by ThreadTwo");
        }
    }
}

class ThreadMain extends Thread {
    public static void main (String args[]) {
        Object objectO = new Object();
        Object objectP = new Object();
         Thread one = new ThreadOne(objectO,
                          Thread.currentThread());
        one.start();
        Thread two = new ThreadTwo(objectO,
                                   objectP);
        two.start();
        try {
            synchronized (objectP)
              { objectP.wait(); } // from HERE!!!
        } catch (Exception e) {}
        synchronized (objectO) {
            objectO.notify();
            System.out.println
              ("\nobjectO.notify by ThreadMain");
        }
    }
}
```

Figure 1. A Multithreaded Java Program with An Asynchronous Exception Throw

P	::=	C^*	program
C	::=	`class` c `ext` c {`var` x^* M^*}	class definition
M	::=	$m(x)$ = e [`throws` c^*]	method definition
id	::=	x	method parameter
	|	id . x	field variable
c			class name
m			method name
x			variable name
e	::=	id	variable
	|	id := e	assignment
	|	`new` c	new object
	|	`this`	self object
	|	e ; e	sequence
	|	`if` e `then` e `else` e	branch
	|	`throw` e	exception raise
	|	`try` e `catch` (c x e)	exception handle
	|	e.m(e)	method call
	|	e.`start`	thread start
	|	e.`wait`	object wait
	|	e.`notify`	object notify
	|	`xthrow` e `to` e	cross exception raise

Figure 2. Abstract Syntax of an Extended Core of Java

that the expressions evaluated after `objectP.wait` in `ThreadMain` may be evaluated concurrently with the `main.stop` in `ThreadOne`.

Section 2 presents the syntax and semantics of source language. Section 3 and Section 4 present two analyses: concurrency analysis and exception analysis, respectively. Section 5 shows related work and Section 6 concludes.

2. Language

For presentation brevity, we consider an imaginary core of Java with the extension of throwing exceptions across threads. Its abstract syntax is in Figure 2. A program is a sequence of class definitions. Class bodies consist of field variable declarations and method definitions. A method definition consists of the method name, its parameter, and its body expression.

Every expression's result is an object. Assignment expression returns the object of its right hand side expression. Sequence expression's result is the object of the last expression in the sequence and the result of a method call is the object from the method body. The `try` expression "`try` e_1 `catch` (c x e_2)" evaluates e_1 first. If the expression returns a normal object then this object is the result of

the `try` expression. If an exception is raised from e_1 and its class is covered by c then the handler expression e_2 is evaluated with the exception object bound to x. If the raised exception is not covered by class c then the raised exception continues to propagate back along the evaluation chain until it meets another handler. Note that nested `try` expression can express multiple handlers for a single expression e_1 : "`try (try` e_1 `catch (`$c_1\ x_1\ e_2$`)) catch (`$c_2\ x_2\ e_3$`)`." The exception object e is raised by `throw` e. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body.

There are three language constructs for multithreading. A new thread e starts by "e`.start`" and the `run` method of the thread e is executed after the thread is started. For the sake of simplicity, we assume that `run` methods are used only for thread start. The `wait` expression "e`.wait`" causes a current thread to wait until the object e is notified by another thread. And `notify` expression "e`.notify`" wakes up some thread which is waiting for the object e to be notified. Note that this waiting and notification mechanism provides support for synchronizing the concurrent executions of threads.

The extended construct for throwing exceptions across threads is "`xthrow` e_1 `to` e_2". It throws the exception object e_1 to the thread e_2. The exception is thrown to the expression evaluated in the thread e_2 at the same time.

We omit the formal semantics of our language. The semantics is not quite different from the existing work [4, 5] which present an event-based structural operational semantics for multi-threaded Java programs.

Throughout this paper, we call `wait`, `notify` expressions and $hd(e_{c.\texttt{run}})$, $e_{c.\texttt{run}}$ for a thread c in the input program *synchronizing expressions* where $hd(e)$ finds the firstly evaluated expression during the evaluation of e. Java threads synchronize with each other by `wait` and `notify` expressions. For a thread c in the input program, the first expression evaluated in c is $hd(e_{c.\texttt{run}})$ and the last expression evaluated is $e_{c.\texttt{run}}$.

3. Step One: Concurrency Analysis

In order to analyze uncaught exceptions in multithreaded Java programs, we need a concurrency analysis which estimates expressions evaluated concurrently in each thread at the same time. According to the semantics of `xthrow` expression, when "`xthrow` e_1 `to` e_2" is evaluated in a thread c_1, the exception e_1 is thrown to the thread e_2. Thus, in order to estimate the expressions in e_2 to which e_1 is thrown, our exception analysis needs to know the expressions in e_2 which are evaluated in parallel with "`xthrow` e_1 `to` e_2".

Section 3.1 presents our approach intuitively. We show the concurrency analysis rules and the soundness of the analysis in Section 3.2 and 3.3.

3.1 Our Approach

The example in Figure 3 (a and b) illustrates how two different executions of the same program result in different synchronizations. The "o_1`.wait`" of Figure 3-(a) synchronizes with "o_1`.notify`" in the thread c_2, while the "o_1`.wait`" of Figure 3-(b) synchronizes with "o_1`.notify`" in the thread c_3. For the former case, since the "o_1`.wait`" is evaluated before e and simultaneously with the "o_1`.notify`" in c_2, all the expressions in c_2 which may be evaluated concurrently with e are included in "the expressions which are evaluated later than "o_1`.notify`" in c_2" ($*_1$). And for the latter case, the "o_1`.wait`" is evaluated before e and simultaneously with the "o_1`.notify`" in c_3, and the "o_2`.notify`" is evaluated before "o_1`.notify`" and simultaneously with the "o_2`.wait`" in c_2. Thus, all the expressions in c_2 which may be evaluated concurrently with e are included in "the expressions which are evaluated later than "o_2`.wait`" in c_2" ($*_2$).

In order to safely estimate the concurrent executions with e in c_2, we consider all the possible synchronizations among threads and, for each synchronization, we collect all the expressions which may be evaluated concurrently with e in c_2. As Figure 3-(c) represents, for each possible synchronization of "o_1`.wait`" with "o_1`.notify`" in c_2 and c_3, our approach collects all the possible concurrently executed expressions $*_1$ and $*_2$.

3.2 Time Analysis Rules

We first analyze timing relation among expressions in a given thread: $pre(e)$ ($suc(e)$) estimates expressions evaluated right before (after) e and $\hat{pre}(e)$ ($\hat{suc}(e)$) estimates the last (next) synchronizing expressions of e.

In Figure 4, every expression e of the input program has two set constraints: $P_e \supseteq se$ and $S_e \supseteq se$. The set variable P_e is for the expressions which may be evaluated right before e and S_e is for the expressions which may be evaluated right after e. A set expression se is either an expression or a union of set expressions. A constraint $P_e \supseteq se$ ($S_e \supseteq se$) may be read as "expression e is evaluated right after (before) one of the expressions in se is evaluated".

Consider the rule for `throw` expression:

$$\frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t \texttt{throw}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}\ .$$

During the evaluation of e, "`throw` e_1", e_1 is firstly evaluated and then e is evaluated. So e is evaluated right after e_1 ($P_e \supseteq e_1$) and e_1 is evaluated right before e ($S_{e_1} \supseteq e$).

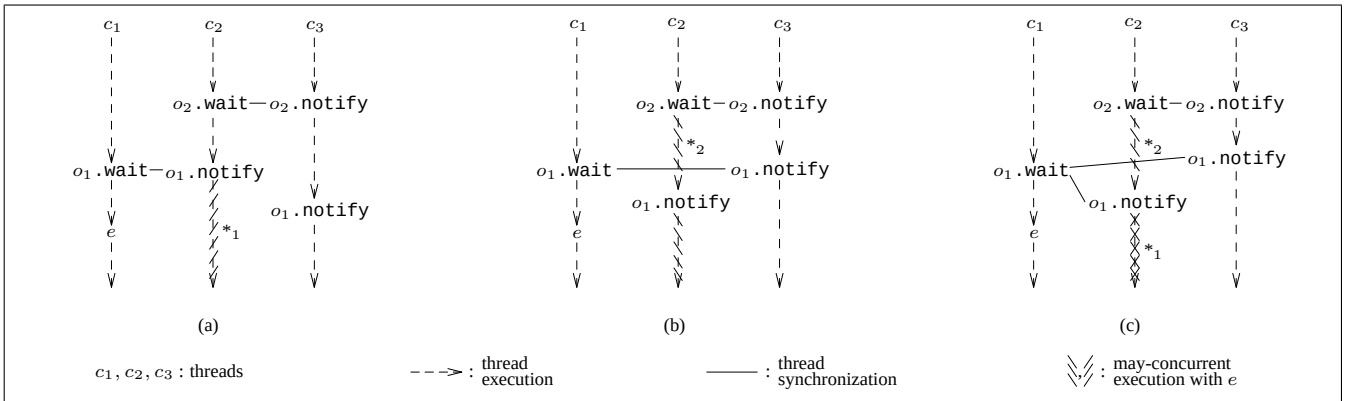

Figure 3. Non-Deterministic Execution Example And Our Approach

Consider the rule for `try` expression:

$$\frac{\triangleright_t e_1 : \mathcal{C}_1 \quad \triangleright_t e_2 : \mathcal{C}_2}{\triangleright_t \texttt{try}\ e_1\ \texttt{catch}\ (c_1\ x_1\ e_2) : \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2, S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

There are two possible evaluations of e, "`try` e_1 `catch` $(c_1\ x_1\ e_2)$": 1. after e_1 is evaluated, e is evaluated when e_1's value is a normal object or e_1's uncaught exceptions are not covered by c_1 2. after e_1 is evaluated, e_2 and then e are evaluated when e_1's uncaught exceptions are covered by c_1. For the former case, e is evaluated right after e_1 ($P_e \supseteq e_1$, $S_{e_1} \supseteq e$). For the latter case, $hd(e_2)$ is evaluated right after e_1 ($P_{hd(e_2)} \supseteq e_1$, $S_{e_1} \supseteq hd(e_2)$) and e is evaluated right after e_2 ($P_e \supseteq e_2$, $S_{e_2} \supseteq e$).

Definition 1 *For a program (a closed expression) $\wp$ and an evaluation of the program ε, $Time_\varepsilon(e)$ denotes the time when e is evaluated in ε. We write $Time(e)$ when it is clear from the context which evaluation the expression e belongs to.*

Our time analysis among expressions safely approximates timing relation among the expressions in a given thread:

Lemma 1 *For a program $\wp$, let $\triangleright_t \wp : \mathcal{C}$ and let $lm(\mathcal{C})$ be the least model of $\mathcal{C}$. If $pre(e) = lm(\mathcal{C})(P_e)$ and $suc(e) = lm(\mathcal{C})(S_e)$, then the followings hold for any evaluation of the program and for any expressions e' and e evaluated in a thread c:*

1. If $Time(e') \leq Time(e)$ then $e' \in pre^*(e)$.
2. If $Time(e') \geq Time(e)$ then $e' \in suc^*(e)$.

Proof. In [15]. □

Another timing analysis $\hat{pre}(e)$ ($\hat{suc}(e)$) which estimates the last (next) synchronizing expressions of an expression in a given thread is omitted for brevity. We refer the interested readers to [15].

3.3 Concurrency Analysis Rules

Now, we present our concurrency analysis rules in this section. We assume that class information $Class(e)$ is already available for every expression e in our analysis. There are several existing work for class analysis[6, 13, 16]. The class analysis estimates for each expression e the classes (including exception classes) that the expression e's normal object belongs to. Note that exception classes are normal classes in Java.

When we say "an expression e in a thread c", it means that e may be evaluated in c. If e is the body of the `run` method of a thread c or e's enclosing method is the `run` method of c, e is evaluated only in c. Otherwise, e is evaluated in threads where e's enclosing method is called. This is formally defined as follows:

$$owner(e) = \begin{cases} \{c\} & \text{where } e = e_c\texttt{.run or } e \in e_c\texttt{.run} \\ \bigcup owner(e_1\texttt{.}m\texttt{(}e_2\texttt{)}) & \\ & \text{where } e = e_m \text{ or } e \in e_m,\ m \neq \texttt{c.run}, \\ & \text{and } e_1\texttt{.}m\texttt{(}e_2\texttt{)} \in \wp. \end{cases}$$

We write $e \in c$ when $c \in owner(e)$.

Figure 5 presents our rules to estimate the concurrency information of a program $\wp$. An edge $e \xrightarrow{c\uparrow c'} e'$ ($e \xrightarrow{c\downarrow c'} e'$) denotes that an expression e' in a thread c' may be evaluated at the same time or before (after) the evaluation of e in c. We show only a half of the rules; dual rules which swap e`.wait` and e`.notify` are omitted for presentation brevity. Rule $[\delta_1]$ is for the case when e in c synchronizes with e' in c' directly. In contrast, rules $[\delta_2]$ to $[\delta_5]$ are for the cases when e in c does not synchronize with e' in c' directly but gets timing relation by using other synchronizations.

Consider rule $[\delta_1]$:

$$\frac{e\texttt{.wait} \in c \quad e_1\texttt{.notify} \in c' \quad Class(e) \cap Class(e_1) \neq \emptyset}{e\texttt{.wait} \xrightarrow{c\uparrow c'} e_1\texttt{.notify} \quad e\texttt{.wait} \xrightarrow{c\downarrow c'} e_1\texttt{.notify}}$$

$$[\text{Program}]_t \quad \frac{\rhd_t C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_t C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{ClassDef}]_t \quad \frac{\rhd_t M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_t \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{MethDef}]_t \quad \frac{\rhd_t e_m : \mathcal{C}}{\rhd_t m(x) = e_m : \mathcal{C}}$$

$$[\text{Variable}]_t \quad \rhd_t id : \emptyset$$

$$[\text{Assign}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t id := e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{New}]_t \quad \rhd_t \texttt{new}\ c : \emptyset$$

$$[\text{This}]_t \quad \rhd_t \texttt{this} : \emptyset$$

$$[\text{Seq}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t e_2 : \mathcal{C}_2}{\rhd_t e_1\ ;\ e_2 : \{\ P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2, S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t e_2 : \mathcal{C}_2 \quad \rhd_t e_3 : \mathcal{C}_3}{\rhd_t \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \{P_{hd(e_2)} \supseteq e_1, P_{hd(e_3)} \supseteq e_1, P_e \supseteq e_2 \cup e_3, S_{e_1} \supseteq hd(e_2) \cup hd(e_3), S_{e_2} \supseteq e, S_{e_3} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\text{Throw}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t \texttt{throw}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t e_2 : \mathcal{C}_2}{\rhd_t \texttt{try}\ e_1\ \texttt{catch}\ (c_1\ x_1\ e_2) : \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2, S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{MethCall}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t e_2 : \mathcal{C}_2}{\rhd_t e_1.m(e_2) : \{P_{hd(e_2)} \supseteq e_1, P_{hd(e_m)} \supseteq e_2, P_e \supseteq e_m, S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq hd(e_m), S_{e_m} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{ThdStart}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t e_1.\texttt{start} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Wait}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t e_1.\texttt{wait} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Notify}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1}{\rhd_t e_1.\texttt{notify} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

$$[\text{Xthrow}]_t \quad \frac{\rhd_t e_1 : \mathcal{C}_1 \quad \rhd_t e_2 : \mathcal{C}_2}{\rhd_t \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2, S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

Figure 4. Time Analysis among Expressions

$$[\delta_1] \quad \frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c' \quad Class(e) \cap Class(e_1) \neq \emptyset}{e.\texttt{wait} \xrightarrow{c \uparrow c'} e_1.\texttt{notify} \quad e.\texttt{wait} \xrightarrow{c \downarrow c'} e_1.\texttt{notify}}$$

$$[\delta_2] \quad \frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \neq c' \quad e_2 \xrightarrow{c'' \uparrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \uparrow c'} e_3}$$

$$[\delta_3] \quad \frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \neq c' \quad e_2 \not\xrightarrow{c'' \uparrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \uparrow c'} hd(e_{c'.\texttt{run}})}$$

$$[\delta_4] \quad \frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \neq c' \quad e_2 \xrightarrow{c'' \downarrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{suc}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \downarrow c'} e_3}$$

$$[\delta_5] \quad \frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \neq c' \quad e_2 \not\xrightarrow{c'' \downarrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{suc}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \downarrow c'} e_{c'.\texttt{run}}}$$

Figure 5. Concurrency Analysis Rules

Since $e.\texttt{wait}$ in c may be notified by $e_1.\texttt{notify}$ in c', these two expressions may be evaluated at the same time.

Consider rule $[\delta_2]$:

$$\frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \xrightarrow{c'' \uparrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \uparrow c'} e_3}.$$

Since $e.\texttt{wait}$ in c may be notified by $e_1.\texttt{notify}$ in c'', these two expressions may be evaluated at the same time. Because e_2 is evaluated before $e_1.\texttt{notify}$ and e_3 may be evaluated at the same time or before e_2, e_3 is evaluated before $e_1.\texttt{notify}$. Thus, e_3 is evaluated before $e.\texttt{wait}$.

Finally, consider rule $[\delta_3]$:

$$\frac{e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \not\xrightarrow{c'' \uparrow c'} e_3 \quad Class(e) \cap Class(e_1) \neq \emptyset \quad e_2 \in \hat{pre}(e_1.\texttt{notify})}{e.\texttt{wait} \xrightarrow{c \uparrow c'} hd(e_{c'.\texttt{run}})}.$$

As the same as in rules $[\delta_1]$ and $[\delta_2]$, $e.\texttt{wait}$ in c and $e_1.\texttt{notify}$ in c'' may be evaluated at the same time. Since the last synchronizing expressions of $e_1.\texttt{notify}$ in c'' may not synchronize with the expressions in c', $e.\texttt{wait}$ cannot get any timing relation here. Thus, $e.\texttt{wait}$ safely makes an edge with $hd(e_{c'.\texttt{run}})$ because $hd(e_{c'.\texttt{run}})$ is the first expression evaluated in c'.

Rules in Figure 5 safely finds the expressions in c' which may be synchronized with the expressions in c before (after) the evaluation of e:

Lemma 2 *For a program $\wp$, let*

$$\delta_{\uparrow}(c, e, c') = \{e' \mid e \xrightarrow{c\uparrow c'} e' \text{ is deducible by the rules in Figure 5}\}$$
$$\delta_{\downarrow}(c, e, c') = \{e' \mid e \xrightarrow{c\downarrow c'} e' \text{ is deducible by the rules in Figure 5}\}.$$

Then the followings hold for any evaluation of the program and an expression e in a thread c in $\wp$:

1. $\exists\, e' \in \delta_{\uparrow}(c, e, c')$ *such that* $Time(e') \leq Time(e)$ *or* $e' = hd(e_{c'.\texttt{run}})$.

2. $\exists\, e' \in \delta_{\downarrow}(c, e, c')$ *such that* $Time(e') \geq Time(e)$ *or* $e' = e_{c'.\texttt{run}}$.

Proof. In [15]. □

Definition 2 *For some threads c, c' in a program $\wp$ and an expression e in c, the concurrency analysis is defined by:*

$$\begin{aligned} Concur(c, e, c') = \{e' \mid\ & e_1 \in \hat{pre}(e),\ e_2 \in \hat{suc}(e), \\ & e'_1 \in \delta_{\uparrow}(c, e_1, c'),\ e'_2 \in \delta_{\downarrow}(c, e_2, c'), \\ & e' \in suc^*(e'_1) \cap pre^*(e'_2)\}. \end{aligned}$$

Our concurrency analysis $Concur(c, e, c') \ni e'$ denotes that e' in the thread c' may be evaluated concurrently with e in c. As Definition 2 shows, the analysis is done by the following steps:

1. Since Java threads synchronize only by `wait` or `notify` expressions, the analysis first finds a pair of the last `wait` or `notify` expressions and the next `wait` or `notify` expressions of e: $e_1 \in \hat{pre}(e)$, $e_2 \in \hat{suc}(e)$.

2. And then the analysis finds expressions in c' which may be synchronized with the expressions in c before the evaluation of e – $e'_1 \in \delta_{\uparrow}(c, e_1, c')$ – and after the evaluation of e – $e'_2 \in \delta_{\downarrow}(c, e_2, c')$.

3. Finally, the analysis collects all the expressions during the pair of synchronized expressions: $e' \in suc^*(e'_1) \cap pre^*(e'_2)$.

Now, we show the soundness of our concurrency analysis:

Theorem 1 *For a program $\wp$ and any evaluation of the program, the expression evaluated in a thread c' concurrently with an expression e in a thread c is included in $Concur(c, e, c')$.*

Proof. In [15]. □

4. Step Two: Exception Analysis

Now, our analysis predicts uncaught exceptions in the input program by using the pre-analyzed concurrency information.

Our exception analysis is presented in the set-constraint framework [9, 3]. As we mentioned before, we assume a safe class information $Class(e)$ for every expression e and we use the safe concurrency information $Concur(c, e, c')$ (Section 3) for every pair of $e \in c$ and c' in the exception analysis.

Figure 6 shows our rules to generate set constraints for the uncaught exception classes from every method and `try`-block of the input program. For each method m and `try`-block e_g in "`try` e_g `catch (`$c\,x$`)`", the set variables X_m and X_g are for the classes that the uncaught exceptions during the evaluation of m's body and e_g, respectively, belong to. A new relation $m \rhd_{\varepsilon} e : \mathcal{C}$ ($g \rhd_{\varepsilon} e : \mathcal{C}$) is read as "a set of constraints $\mathcal{C}$ is generated from an expression e which is a subexpression of the body of a method m (`try`-block e_g)."

Consider the rule for `throw` expression:

$$\frac{m \rhd_{\varepsilon}\ e_1 : \mathcal{C}_1}{m \rhd_{\varepsilon}\ \texttt{throw}\, e_1 : \{X_m \supseteq Class(e_1)\} \cup \mathcal{C}_1}.$$

Uncaught exceptions from the method m, X_m, include the exception classes of the expression e_1 ($Class(e_1)$).

Consider the rule for `try` expression:

$$\frac{g \rhd_{\varepsilon} e_g : \mathcal{C}_g\ \ m \rhd_{\varepsilon} e_1 : \mathcal{C}_1}{\begin{array}{l} m \rhd_{\varepsilon} \texttt{try}\, e_g\, \texttt{catch}\, (c_1\, x_1\, e_1) : \\ \qquad \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup\ \mathcal{C}_g \cup \mathcal{C}_1 \end{array}}.$$

Among the classes of raised exceptions from e_g, those exceptions covered by c_1 are not included in the uncaught exceptions from m. Note that the left-hand side of the derivation rule for e_g is g.

Finally, consider the rule for `xthrow` expression:

$$\frac{m \rhd_{\varepsilon} e_1 : \mathcal{C}_1\ \ m \rhd_{\varepsilon} e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd_{\varepsilon} \texttt{xthrow}\, e_1\, \texttt{to}\, e_2 : \\ \qquad \{X_{m'} \supseteq Class(e_1) \mid c \in owner(\texttt{xthrow}\, e_1\, \texttt{to}\, e_2), \\ \qquad\qquad c' \in Class(e_2), m'(x) = e_{m'} \in \wp, e' \in e_{m'}, \\ \qquad\qquad e' \in Concur(c, \texttt{xthrow}\, e_1\, \texttt{to}\, e_2, c')\} \\ \qquad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}.$$

Exceptions e_1 are thrown to the methods m' ($X_{m'} \supseteq Class(e_1)$) whose subexpressions e' may be evaluated in the thread $c' \in Class(e_2)$ concurrently with the evaluation of `xthrow` expression in the thread c ($Concur(c, \texttt{xthrow}\, e_1\, \texttt{to}\, e_2, c')$).

Our exception analysis is safe:

$$[\text{Program}]_\varepsilon \quad \frac{\rhd_\varepsilon C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_\varepsilon C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{ClassDef}]_\varepsilon \quad \frac{\rhd_x M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_x \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

$$[\text{MethDef}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_m : \mathcal{C}}{\rhd_\varepsilon m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m : \mathcal{C}}$$

$$[\text{Variable}]_\varepsilon \quad m \rhd_\varepsilon id : \emptyset$$

$$[\text{Assign}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon id\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

$$[\text{New}]_\varepsilon \quad m \rhd_\varepsilon \texttt{new}\ c : \emptyset$$

$$[\text{This}]_\varepsilon \quad m \rhd_\varepsilon \texttt{this} : \emptyset$$

$$[\text{Seq}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1 \quad m \rhd_\varepsilon e_2 : \mathcal{C}_2}{m \rhd_\varepsilon e_1\ \texttt{;}\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{Cond}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1 \quad m \rhd_\varepsilon e_2 : \mathcal{C}_2 \quad m \rhd_\varepsilon e_3 : \mathcal{C}_3}{m \rhd_\varepsilon \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

$$[\text{Throw}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon \texttt{throw}\ e_1 : \{X_m \supseteq \mathit{Class}(e_1)\} \cup \mathcal{C}_1}$$

$$[\text{Try}]_\varepsilon \quad \frac{g \rhd_\varepsilon e_g : \mathcal{C}_g \quad m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon \texttt{try}\ e_g\ \texttt{catch}\ (c_1\ x_1\ e_1) : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

$$[\text{MethCall}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1 \quad m \rhd_\varepsilon e_2 : \mathcal{C}_2}{m \rhd_\varepsilon e_1\texttt{.}m'\texttt{(}e_2\texttt{)} : \{X_m \supseteq X_{c'.m'} \mid c' \in \mathit{Class}(e_1), m'(x) = e_{m'} \in c'\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

$$[\text{ThdStart}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon e_1\texttt{.start} : \mathcal{C}_1}$$

$$[\text{Wait}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon e_1\texttt{.wait} : \mathcal{C}_1}$$

$$[\text{Notify}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1}{m \rhd_\varepsilon e_1\texttt{.notify} : \mathcal{C}_1}$$

$$[\text{Xthrow}]_\varepsilon \quad \frac{m \rhd_\varepsilon e_1 : \mathcal{C}_1 \quad m \rhd_\varepsilon e_2 : \mathcal{C}_2}{m \rhd_\varepsilon \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{\, X_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), c' \in \mathit{Class}(e_2), e' \in e_{m'}, m'(x) = e_{m'} \in \wp, e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

Figure 6. Exception Analysis at Method-Level

Theorem 2 *For a program $\wp$, let $\rhd_\varepsilon \wp : \mathcal{C}$. Then, if $m \rhd_\varepsilon e : \mathcal{C}'$ occurs during $\rhd_\varepsilon \wp : \mathcal{C}$ then $lm(\mathcal{C})(X_m)$ includes all the exceptions that escape from e during the execution of $\wp$.*

Proof. In [15]. □

5. Related Work

Several exception analyses have been developed to detect uncaught exceptions in ML programs [17, 19, 20, 12]. Yi first designed an exception analysis for Standard ML (SML) programs based on abstract interpretation [17], which was very accurate but too slow. Yi and Ryu redesigned the analysis by set-based framework [19, 20], which shows a good cost-accuracy performance. Fähndrich and Aiken [7] developed an exception analyzer for SML programs by using their BANE (Berkeley ANalysis Engine) toolkit. As in effect systems, their system casts the type and exception inference. Leroy and Pessaux [12] developed an exception analysis for OCaml programs which relies on a unification-based type inference in a non-standard type system. Their type system uses unified mechanisms both to collect the sets of uncaught exceptions of expressions and to refine the usual ML types by more precise information about the possible values of expressions.

Even though Java compiler ensures the programmer's specification for the possible uncaught exceptions in each method definition, there exist several work for helping fine-grained exception handling. Yi and Chang [18] developed an exception analyzer for single-threaded Java programs by a set-constraint framework, which estimated the exception flows independently of the programmer's specifications. Robillard and Murphy [14] developed a tool called Jex that analyzes the flow of exceptions in Java programs by analyzing exception handling expressions, but Jex does not report asynchronous exceptions.

Since Java does not provide a good method for multi-threaded exception handling, several work were proposed to deal with the problem. Lea [11] presented a method called *Completion Callbacks*. In order for a thread A to know whether the other thread B completed successfully or with an exception, A should implement a predefined interface and B must call this interface to indicate success or failure. Similarly, Haggar [10] proposed a solution by using the listener paradigm. If a thread A wants to get an asynchronous signal from a thread B, A should register as a listener of B and notification is sent to objects that are registered as listeners of B by calling a predefined method.

6. Conclusion

We have presented an exception analysis for multi-threaded Java programs. In Java, throwing exceptions

across threads is deprecated because of the safety problem. Instead of restricting programmers' freedom, we extend Java language to support multithreaded exception handling and propose a tool to detect uncaught exceptions in the input programs. Our analysis firstly estimates concurrently evaluated expressions among threads, and then predicts uncaught exceptions by using the pre-analyzed concurrency information.

Our method can be applied to other languages with both exception handling facilities and multithreading mechanisms. One contribution of this paper is an analysis in divide and conquer style. By dividing one complex analysis to several simpler analyses, the analysis result is achieved systematically. Another merit of our method is designing a sparse constraint system. When the interesting properties are sparse in programs, it is reasonable to analyze the properties at a larger granularity than at every expression.

References

[1] Java™ 2 SDK, Standard Edition 1.2. `http://java.sun.com/products/jdk/1.2`.

[2] Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated? `http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html`.

[3] A. Aiken and N. Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, Jan. 1995.

[4] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From sequential to multi-threaded java: An event-based operational semantics. In M. Johnson, editor, *The Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology*, volume 1349 of *Lecture Notes in Computer Science*, pages 75–90. Springer–Verlag, 1997.

[5] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer–Verlag, 1999.

[6] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, Jan. 1998.

[7] M. Fähndrich, J. S. Foster, A. Aiken, and J. Cu. Tracking down exceptions in standard ml programs. Technical Report UCB/CSD-98-996, Computer Science Division, University of California, Berkeley, Feb. 1998.

[8] J. Gosling, B. Joy, G. L. S. Jr., and G. Bracha. *The Java™ Language Specification*. Addison Wesley, second edition edition, 2000.

[9] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie Mellon University, Oct. 1992.

[10] J. D. R. III and P. Haggar. Multithreaded exception handling in java. Java Report, Aug. 1998.

[11] D. Lea. *Concurrent Programming in Java™, Second Edition: Design Principles and Patterns*. Addison-Wesley, 1999.

[12] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, Mar. 2000.

[13] J. Palsberg and M. I. Schwarzbach. Object-oriented type inference. In *Proceedings of ACM Conference on OOPSLA*, pages 141–161, 1991.

[14] M. P. Robillard and G. C. Murphy. Analyzing exception flow in java programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337. Springer-Verlag, 1999.

[15] S. Ryu. Exception analysis for multithreaded Java programs. Technical Memorandum ROPAS-2001-11, Research On Program Analysis System, Korea Advanced Institute of Science and Technology, Apr. 2001.

[16] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of ACM Conference on OOPSLA*, pages 281–293, 2000.

[17] K. Yi. Compile-time detection of uncaught exceptions for Standard ML programs. In *Lecture Notes in Computer Science*, volume 864, pages 238–254. Springer-Verlag, proceedings of the first international static analysis symposium edition, 1994.

[18] K. Yi and B.-M. Chang. Exception analysis for java. In A. Moreira and D. Demeyer, editors, *Object-Oriented Technology. ECOOP'99 Workshop Reader (Formal Techniques for Java Programs)*, volume 1743 of *Lecture Notes in Computer Science*, pages 111–112. Springer-Verlag, June 1999.

[19] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In *Proceedings of the 4th International Static Analysis Symposium*, volume 1302 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 1997.

[20] K. Yi and S. Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Extended version of [19] (to appear).