Research On Program Analysis System
National Creative Research Initiative Center
ROPAS Korea Advanced Institute of Science and Technology

ROPAS MEMO
2001-11
April 20, 2001

Exception Analysis for Multithreaded Java Programs*

Sukyoung Ryu
puppy@ropas.kaist.ac.kr

April 20, 2001

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 [GJJB00] 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 [JDK] because it is unsafe [Dep]:

> "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

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

```java
private Thread blinker;

public void start() {
  blinker = new Thread(this);
  blinker.start();
}

public void stop() {
  blinker.stop();         // UNSAFE!
}

public void run() {
  Thread thisThread =
      Thread.currentThread();
  while (true) {
    try {
      thisThread.sleep(interval);
    } catch
      (InterruptedException e) {
    }
    repaint();
  }
}
```

```java
private volatile Thread blinker;




public void stop() {
  blinker = null;
}

public void run() {
  Thread thisThread =
      Thread.currentThread();
  while (blinker == thisThread) {
    try {
      thisThread.sleep(interval);
    } catch
      (InterruptedException e) {
    }
    repaint();
  }
}
```

Figure 1: Example[1] with/without `Thread.stop`

stop if the flag is set. Consider the example codes in Figure 1. The use of `blinker.stop` in the left example is replaced with checking whether the variable `blinker` is `null` in the right example.

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 2. 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 the

[1] This example is from [Dep]

```
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) {}
        main.stop(new Exception());                    // asynchronous exception throw
    }
}

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 2: A Multithreaded Java Program with An Asynchronous Exception Throw

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

Figure 3: Abstract Syntax of an Extended Core of Java

threads `ThreadMain`, `ThreadOne` and `ThreadTwo`, our concurrency analysis estimates 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 describes the analysis complexity, Section 6 shows related work, and Section 7 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 3. 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 [CKRW97, CKRW99] which present an event-based structural operational semantics for multi-threaded Java programs.

2.1 Synchronizing Expressions

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 $hd(e)$ is defined as follows:

$$
\begin{array}{rlcl}
hd(e) = & \text{case } e \text{ of} & & \\
& id & \Rightarrow & e \\
& id \texttt{ := } e_1 & \Rightarrow & hd(e_1) \\
& \texttt{new}\, c & \Rightarrow & e \\
& \texttt{this} & \Rightarrow & e \\
& e_1 \texttt{ ; } e_2 & \Rightarrow & hd(e_1) \\
& \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \Rightarrow & hd(e_1) \\
& \texttt{throw}\, e_1 & \Rightarrow & hd(e_1) \\
& \texttt{try } e_1 \texttt{ catch } (c_1\ x_1\ e_2) & \Rightarrow & hd(e_1) \\
& e_1\texttt{.}m(e_2) & \Rightarrow & hd(e_1) \\
& e_1\texttt{.start} & \Rightarrow & hd(e_1) \\
& e_1\texttt{.wait} & \Rightarrow & hd(e_1) \\
& e_1\texttt{.notify} & \Rightarrow & hd(e_1) \\
& \texttt{xthrow}\, e_1 \texttt{ to } e_2 & \Rightarrow & hd(e_1).
\end{array}
$$

The auxiliary function $hd(e)$ 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 $e_{c.\texttt{run}}$.

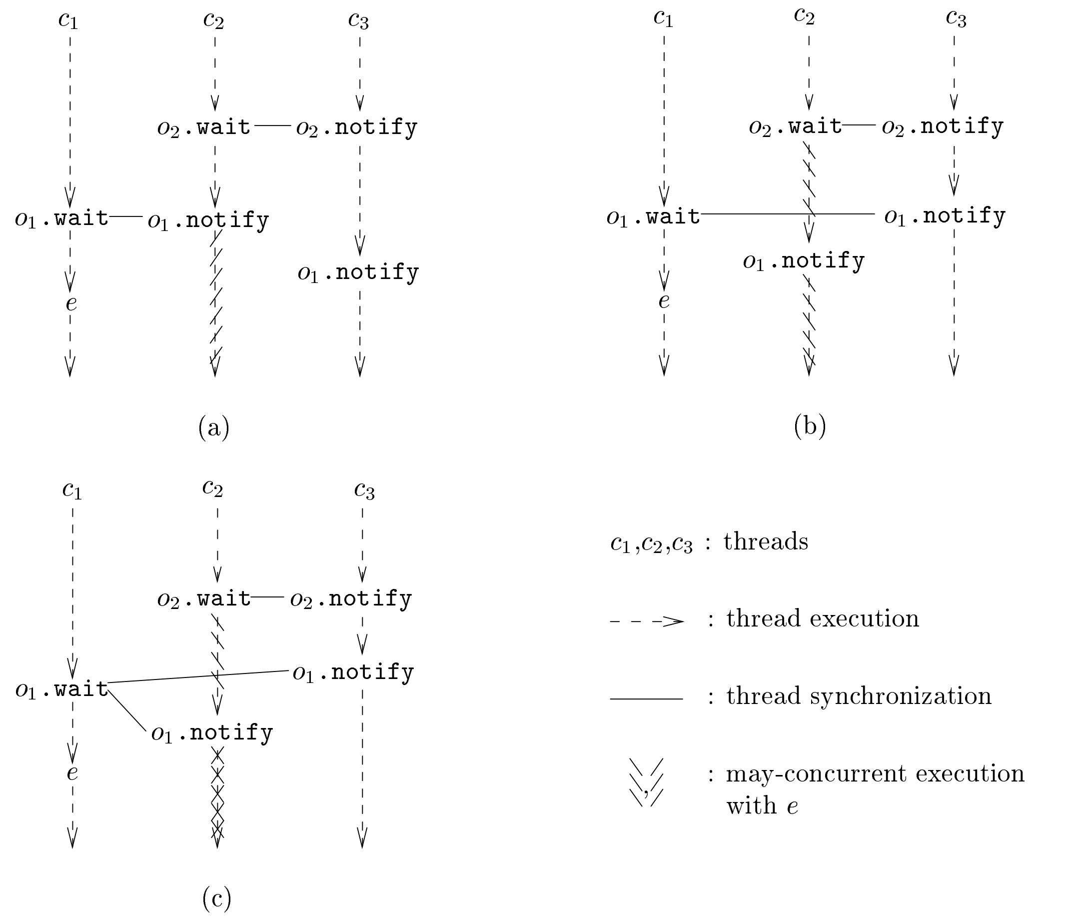

Figure 4: Non-Deterministic Execution Example And Our Approach

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 prove the soundness of the analysis in Section 3.2 and 3.3.

3.1 Our Approach

The example in Figure 4 (a and b) illustrates how two different executions of the same program result in different synchronizations. The "o_1`.wait`" of Figure 4-(a) synchronizes with "o_1`.notify`" in the thread c_2, while the "o_1`.wait`" of Figure 4-

(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 4-(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. In Section 3.2.1, $pre(e)$ ($suc(e)$) estimates expressions evaluated right before (after) e and $\hat{pre}(e)$ ($\hat{suc}(e)$) estimates the last (next) synchronizing expressions of e in Section 3.2.2.

3.2.1 Time Analysis among Expressions

In Figure 5, 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 ($S_{e_1} \supseteq e$).

Consider the rule for `try` expression:

$$\frac{\rhd_t\, e_1 : \mathcal{C}_1 \quad \rhd_t\, e_2 : \mathcal{C}_2}{\rhd_t\, \texttt{try}\ e_1\ \texttt{catch}\ (c_1\ x_1\ e_2) : \begin{array}{l}\{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2,\\ \;S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}\ .$$

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 ($P_e \supseteq e_2$, $S_{e_2} \supseteq e$).

$[\text{Program}]_t$	$\dfrac{\triangleright_t\ C_i : \mathcal{C}_i,\ i = 1,\cdots,n}{\triangleright_t\ C_1,\cdots,C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$
$[\text{ClassDef}]_t$	$\dfrac{\triangleright_t\ M_i : \mathcal{C}_i,\ i = 1,\cdots,n}{\triangleright_t\ \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$
$[\text{MethDef}]_t$	$\dfrac{\triangleright_t\ e_m : \mathcal{C}}{\triangleright_t\ m(x)\ \texttt{=}\ e_m : \mathcal{C}}$
$[\text{Variable}]_t$	$\triangleright_t\ id : \emptyset$
$[\text{Assign}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1}{\triangleright_t\ id\ \texttt{:=}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$
$[\text{New}]_t$	$\triangleright_t\ \texttt{new}\ c : \emptyset$
$[\text{This}]_t$	$\triangleright_t\ \texttt{this} : \emptyset$
$[\text{Seq}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1 \quad \triangleright_t\ e_2 : \mathcal{C}_2}{\triangleright_t\ e_1\ \texttt{;}\ e_2 : \{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2, S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}$
$[\text{Cond}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1 \quad \triangleright_t\ e_2 : \mathcal{C}_2 \quad \triangleright_t\ e_3 : \mathcal{C}_3}{\triangleright_t\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \begin{array}{l}\{P_{hd(e_2)} \supseteq e_1, P_{hd(e_3)} \supseteq e_1, P_e \supseteq e_2 \cup e_3,\\ \quad S_{e_1} \supseteq hd(e_2) \cup hd(e_3), S_{e_2} \supseteq e, S_{e_3} \supseteq e\}\\ \cup\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3\end{array}}$
$[\text{Throw}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1}{\triangleright_t\ \texttt{throw}\ e_1 : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$
$[\text{Try}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1 \quad \triangleright_t\ e_2 : \mathcal{C}_2}{\triangleright_t\ \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \begin{array}{l}\{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_1 \cup e_2,\\ \quad S_{e_1} \supseteq hd(e_2) \cup e, S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$
$[\text{MethCall}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1 \quad \triangleright_t\ e_2 : \mathcal{C}_2}{\triangleright_t\ e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \begin{array}{l}\{P_{hd(e_2)} \supseteq e_1, P_{hd(e_m)} \supseteq e_2, P_e \supseteq e_m,\\ \quad S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq hd(e_m), S_{e_m} \supseteq e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$
$[\text{ThdStart}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1}{\triangleright_t\ e_1\texttt{.start} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$
$[\text{Wait}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1}{\triangleright_t\ e_1\texttt{.wait} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$
$[\text{Notify}]_t$	$\dfrac{\triangleright_t\ e_1 : \mathcal{C}_1}{\triangleright_t\ e_1\texttt{.notify} : \{P_e \supseteq e_1, S_{e_1} \supseteq e\} \cup \mathcal{C}_1}$
$[\text{Xthrow}]_t$	$\dfrac{\triangleright_t e_1 : \mathcal{C}_1 \quad \triangleright_t\ e_2 : \mathcal{C}_2}{\triangleright_t\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \begin{array}{l}\{P_{hd(e_2)} \supseteq e_1, P_e \supseteq e_2,\\ \quad S_{e_1} \supseteq hd(e_2), S_{e_2} \supseteq e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$

Figure 5: Time Analysis among Expressions

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 $e' \in suc^*(e)$.*

The proof uses Lemma 2, Lemma 3, and the auxiliary function $ExSq(e)$ which represents a set of all the possible execution sequences of e as follows:

$$
\begin{array}{lll}
ExSq(e) = & \text{case } e \text{ of} & \\
& id & \Rightarrow \{e\} \\
& id\ \texttt{:=}\ e_1 & \Rightarrow ExSq(e_1) \oplus \{e\} \\
& \texttt{new}\ c & \Rightarrow \{e\} \\
& \texttt{this} & \Rightarrow \{e\} \\
& e_1\ \texttt{;}\ e_2 & \Rightarrow ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\} \\
& \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \Rightarrow ExSq(e_1) \oplus (ExSq(e_2) \cup ExSq(e_3)) \oplus \{e\} \\
& \texttt{throw}\ e_1 & \Rightarrow ExSq(e_1) \oplus \{e\} \\
& \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} & \Rightarrow (ExSq(e_1) \cup (ExSq(e_1) \oplus ExSq(e_2))) \oplus \{e\} \\
& e_1\texttt{.}m\texttt{(}e_2\texttt{)} & \Rightarrow ExSq(e_1) \oplus ExSq(e_2) \oplus ExSq(e_m) \oplus \{e\} \\
& & \quad \text{where } c \in Class(e_1),\ m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m \in c \\
& e_1\texttt{.start} & \Rightarrow ExSq(e_1) \oplus \{e\} \\
& e_1\texttt{.wait} & \Rightarrow ExSq(e_1) \oplus \{e\} \\
& e_1\texttt{.notify} & \Rightarrow ExSq(e_1) \oplus \{e\} \\
& \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 & \Rightarrow ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}
\end{array}
$$

$$\{e_1e_2,\ e_3e_4\} \oplus \{e_5e_6,\ e_7e_8\} = \{e_1e_2e_5e_6,\ e_3e_4e_5e_6,\ e_1e_2e_7e_8,\ e_3e_4e_7e_8\}$$

Note that for any evaluation of an expression e, $e_1 \cdots e_n \in ExSq(e)$, if $Time(e_i) \leq Time(e_j)$ $(1 \leq i, j \leq n)$ then $1 \leq i \leq j \leq n$.

Lemma 2 *For a program $\wp$, the followings hold for a thread c in $\wp$ and any evaluation $e_1 \cdots e_n \in ExSq(e_c\texttt{.run})$ of the thread:*

1. *$e_i \in pre(e_{i+1}) \quad (1 \leq i < n)$*
2. *$e_{i+1} \in suc(e_i) \quad (1 \leq i < n)$*

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. The execution sequence "e_ie_{i+1}" is generated by either $ExSq(e') \oplus \{e_{i+1}\}$ $(e \cdots e_i \in ExSq(e'))$ or $ExSq(e') \oplus ExSq(e'')$ $(e'_1 \cdots e_i \in ExSq(e'),\ e_{i+1} \cdots e'_2 \in ExSq(e''))$. Note that for any $e_1 \cdots e_n \in ExSq(e)$, $e_1 = hd(e)$ and $e_n = e$. Let $first(ExSq(e)) = hd(e)$, $first(\{e\}) = e$, and $last(ExSq(e)) = e$.

We prove by the case analysis of the execution sequence $e_i e_{i+1}$. For each case of $S_1 \oplus S_2$, we need to show that if $\mathit{last}(S_1) = e_i$ and $\mathit{first}(S_2) = e_{i+1}$, then $e_i \in \mathit{lm}(\mathcal{C})(P_{e_{i+1}})$. And it is sufficient to show that, for each $S_1 \oplus S_2$, if $\mathit{last}(S_1) = e_i$ and $\mathit{first}(S_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$.

- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = \mathit{id}$ `:=` e_1
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Assign}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e = e_1$ `;` e_2
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Seq}]_t$.
- `Case` $\mathit{ExSq}(e_2) \oplus \{e\}$ where $e = e_1$ `;` e_2
  If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Seq}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e =$ `if` e_1 `then` e_2 `else` e_3
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Cond}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_3)$ where $e =$ `if` e_1 `then` e_2 `else` e_3
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_3)) = \mathit{hd}(e_3) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Cond}]_t$.
- `Case` $\mathit{ExSq}(e_2) \oplus \{e\}$ where $e =$ `if` e_1 `then` e_2 `else` e_3
  If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Cond}]_t$.
- `Case` $\mathit{ExSq}(e_3) \oplus \{e\}$ where $e =$ `if` e_1 `then` e_2 `else` e_3
  If $\mathit{last}(\mathit{ExSq}(e_3)) = e_3 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Cond}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e =$ `throw` e_1
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Throw}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e =$ `try` e_1 `catch` `(`c_1 x_1 e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Try}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e =$ `try` e_1 `catch` `(`c_1 x_1 e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Try}]_t$.
- `Case` $\mathit{ExSq}(e_2) \oplus \{e\}$ where $e =$ `try` e_1 `catch` `(`c_1 x_1 e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Try}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \mathit{ExSq}(e_2)$ where $e = e_1$`.`m`(`e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_2)) = \mathit{hd}(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{MethCall}]_t$.
- `Case` $\mathit{ExSq}(e_2) \oplus \mathit{ExSq}(e_m)$ where $e = e_1$`.`m`(`e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_2)) = e_2 = e_i$ and $\mathit{first}(\mathit{ExSq}(e_m)) = \mathit{hd}(e_m) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{MethCall}]_t$.
- `Case` $\mathit{ExSq}(e_m) \oplus \{e\}$ where $e = e_1$`.`m`(`e_2`)`
  If $\mathit{last}(\mathit{ExSq}(e_m)) = e_m = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{MethCall}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = e_1$`.start`
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{ThdStart}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = e_1$`.wait`
  If $\mathit{last}(\mathit{ExSq}(e_1)) = e_1 = e_i$ and $\mathit{first}(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\mathrm{Wait}]_t$.
- `Case` $\mathit{ExSq}(e_1) \oplus \{e\}$ where $e = e_1$`.notify`

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Notify}]_t$.

– `Case` $ExSq(e_1) \oplus ExSq(e_2)$ where $e = \texttt{xthrow}\ e_1\ \texttt{to}\ e_2$

If $last(ExSq(e_1)) = e_1 = e_i$ and $first(ExSq(e_2)) = hd(e_2) = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Xthrow}]_t$.

– `Case` $ExSq(e_2) \oplus \{e\}$ where $e = \texttt{xthrow}\ e_1\ \texttt{to}\ e_2$

If $last(ExSq(e_2)) = e_2 = e_i$ and $first(\{e\}) = e = e_{i+1}$, then $P_{e_{i+1}} \supseteq e_i$ by $[\text{Xthrow}]_t$.

□

Lemma 3 *For a program $\wp$, the followings hold for any evaluation of the program:*

1. *If $e_1 \in pre(e_2)$ then $pre^*(e_1) \subseteq pre^*(e_2)$.*
2. *If $e_1 \in suc(e_2)$ then $suc^*(e_1) \subseteq suc^*(e_2)$.*

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. First, we prove that if $e_1 \in pre(e_2)$ then $pre^n(e_1) \subseteq pre^{n+1}(e_2)$ for any n. We prove by induction on n.

– `Case` $n = 1$: $pre^2(e_2) = \bigcup_{e \in pre(e_2)} pre(e) \supseteq pre(e_1)$.

– `Case` $n = k$:

$$\begin{aligned} pre^{k+1}(e_2) &= \bigcup_{e \in pre^k(e_2)} pre(e) \\ &\supseteq \bigcup_{e \in pre^{k-1}(e_1)} pre(e) && \text{by I.H.} \\ &= pre^k(e_1). \end{aligned}$$

Since the set of expressions in the input program is finite, if $e_1 \in pre(e_2)$ then $pre^*(e_1) \subseteq pre^*(e_2)$. □

Proof of Lemma 1. Since the second property is a dual of the first one, we only prove the first property here.

1. We need to show that

$$\forall c \in \wp.\ e_1 \cdots e_n \in ExSq(e_c.\texttt{run}).\ 1 \leq i \leq j \leq n \Rightarrow e_i \in pre^*(e_j).$$

We prove by induction on $j - i$.

–`Case` $j - i = 0$: $e_i \in pre^*(e_i) = pre^*(e_j)$.

–`Case` $j - i = m$: We need to show that $e_i \in pre^*(e_{i+m})$. By I.H., $e_i \in pre^*(e_{i+m-1})$ and $e_{i+m-1} \in pre(e_{i+m})$ by Lemma 2. Then by Lemma 3, $pre^*(e_{i+m-1}) \subseteq pre^*(e_{i+m})$. Thus, $e_i \in pre^*(e_{i+m-1}) \subseteq pre^*(e_{i+m})$. □

3.2.2 Time Analysis among Synchronizing Expressions

We present another timing analysis which estimates the last and next synchronizing expressions of an expression in a given thread. In Figure 6, every expression e of the input program has two set constraints: $\hat{P}_e \supseteq se$ and $\hat{S}_e \supseteq se$. The set variable $\hat{P}_e$ is for the synchronizing expressions which are to be evaluated before e and $\hat{S}_e$ is for the synchronizing expressions which are to be evaluated after e. A set expression se is an expression, a set variable, or a union of set expressions. A constraint $\hat{P}_e \supseteq se$ ($\hat{S}_e \supseteq se$) may be read as "synchronizing expressions se are evaluated before (after) the expression e is evaluated".

$$p(e) = \begin{cases} \{e\} & \text{if } e \text{ is a synchronizing expression} \\ \hat{P}_e & \text{otherwise} \end{cases} \qquad s(e) = \begin{cases} \{e\} & \text{if } e \text{ is a synchronizing expression} \\ \hat{S}_e & \text{otherwise} \end{cases}$$

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

[ClassDef]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_{\hat{t}}\ \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[MethDef1]$_{\hat{t}}$
$$\frac{e_m : \mathcal{C} \quad m \neq \texttt{run}}{\rhd_{\hat{t}}\ m(x)\ \texttt{=}\ e_m : \mathcal{C}}$$

[MethDef2]$_{\hat{t}}$
$$\frac{e_m : \mathcal{C} \qquad m = \texttt{run}}{\rhd_{\hat{t}}\ m(x)\ \texttt{=}\ e_m : \{\hat{P}_{hd(e_m)} \supseteq hd(e_m), \hat{S}_{e_m} \supseteq e_m\} \cup \mathcal{C}}$$

[Variable]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ id : \emptyset$$

[Assign]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ id\ \texttt{:=}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[New]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ \texttt{new}\ c : \emptyset$$

[This]$_{\hat{t}}$
$$\rhd_{\hat{t}}\ \texttt{this} : \emptyset$$

[Seq]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\rhd_{\hat{t}}\ e_1\ \texttt{;}\ e_2 : \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_2), \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2 \quad \rhd_{\hat{t}}\ e_3 : \mathcal{C}_3}{\begin{array}{l} \rhd_{\hat{t}}\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \ \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_{hd(e_3)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_1), \\ \qquad\qquad \hat{S}_{e_1} \supseteq \hat{S}_e, \hat{S}_{e_2} \supseteq \hat{S}_e, \hat{S}_{e_3} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \end{array}}$$

[Throw]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ \texttt{throw}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[Try]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{l} \rhd_{\hat{t}}\texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \ \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_1), \\ \qquad\qquad \hat{S}_{e_1} \supseteq \hat{S}_e, \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

[MethCall]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{l} \rhd_{\hat{t}}\ e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \ \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_{hd(e_m)} \supseteq p(e_2), \hat{P}_e \supseteq p(e_m), \\ \qquad \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq s(hd(e_m)), \hat{S}_{e_m} \supseteq \hat{S}_e\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

[ThdStart]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.start} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}$$

[Wait]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.wait} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

[Notify]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}}\ e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\ e_1\texttt{.notify} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1}$$

[Xthrow]$_{\hat{t}}$
$$\frac{\rhd_{\hat{t}} e_1 : \mathcal{C}_1 \quad \rhd_{\hat{t}}\ e_2 : \mathcal{C}_2}{\begin{array}{l} \rhd_{\hat{t}}\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \ \{\hat{P}_{hd(e_2)} \supseteq p(e_1), \hat{P}_e \supseteq p(e_2), \\ \qquad \hat{S}_{e_1} \supseteq s(hd(e_2)), \hat{S}_{e_2} \supseteq \hat{S}_e\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 6: Time Analysis among Synchronizing Expressions

Consider the `throw` expression:

$$\frac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, \texttt{throw}\ e_1 : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq \hat{S}_e\} \cup \mathcal{C}_1}\ .$$

If e_1 is either `wait` or `notify` expression, e_1 is the last synchronizing expression of e ($\hat{P}_e \supseteq e_1$). Otherwise, the last synchronizing expressions of e includes those of e_1 ($\hat{P}_e \supseteq \hat{P}_{e_1}$). Since e is not a synchronizing expression, the next synchronizing expressions of e_1 includes those of e ($\hat{S}_{e_1} \supseteq \hat{S}_e$).

Consider the `wait` and `notify` expressions:

$$\frac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, e_1\texttt{.wait} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1} \qquad \frac{\rhd_{\hat{t}}\, e_1 : \mathcal{C}_1}{\rhd_{\hat{t}}\, e_1\texttt{.notify} : \{\hat{P}_e \supseteq p(e_1), \hat{S}_{e_1} \supseteq e\} \cup \mathcal{C}_1}\ .$$

The last synchronizing expressions of e is either e_1 ($\hat{P}_e \supseteq e_1$) if e_1 is `wait` or `notify` expression, or those of e_1 ($\hat{P}_e \supseteq \hat{P}_{e_1}$) otherwise. Since e is a synchronizing expression, the next synchronizing expressions of e_1 includes e ($\hat{S}_{e_1} \supseteq e$).

There are two rules for method definitions:

$$\frac{e_m : \mathcal{C} \quad m \neq \texttt{run}}{\rhd_{\hat{t}}\, m(x)\ \texttt{=}\ e_m : \mathcal{C}} \qquad \frac{e_m : \mathcal{C} \quad m = \texttt{run}}{\rhd_{\hat{t}}\, m(x)\ \texttt{=}\ e_m : \{\hat{P}_{hd(e_m)} \supseteq hd(e_m), \hat{S}_{e_m} \supseteq e_m\} \cup \mathcal{C}}\ .$$

For normal methods, the analysis does not generate any constraint. In case of `run` methods, $\hat{P}_{hd(e_m)} \supseteq hd(e_m)$ and $\hat{S}_{e_m} \supseteq e_m$ are generated since $hd(e_{\texttt{run}})$ and $e_{\texttt{run}}$ are synchronizing expressions.

Our second time analysis lower approximates timing relation among synchronizing expressions in a given thread:

Lemma 4 *For a program $\wp$, let $\rhd_{\hat{t}}\wp : \mathcal{C}$ and let $lm(\mathcal{C})$ be the least model of $\mathcal{C}$. If $\hat{pre}(e) = lm(\mathcal{C})(\hat{P}_e)$ and $\hat{suc}(e) = lm(\mathcal{C})(\hat{S}_e)$, then the followings hold for any evaluation of the program and for any expressions e evaluated in a thread c:*

1. $\forall e' \in \hat{pre}(e).\ Time(e') \leq Time(e)$.
2. $\forall e' \in \hat{suc}(e).\ Time(e') \geq Time(e)$.

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. The least model $lm(\mathcal{C})$ is equivalent to the $\subseteq$-least fixpoint fix$\mathcal{F}$ of the following

continuous function $\mathcal{F}$ derived from $\mathcal{C}$ as follows[CC95]:

$$
\mathcal{F}(\rho)(\hat{P}_e) = \begin{cases}
\{e_1\} & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\},\ e \neq hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp, \\ e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\rho(\hat{P}_{e_1}) & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\},\ e \neq hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp, \\ e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\{e_1, e''\} & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\ e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is a synch. exp.,} \\ e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\{e_1\} \cup \rho(\hat{P}_{e''}) & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\ e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is not a synch. exp.,} \\ e = hd(e_2) \text{ and } e_1 \text{ is a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\rho(\hat{P}_{e_1}) \cup \{e''\} & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\ e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is a synch. exp.,} \\ e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\rho(\hat{P}_{e_1}) \cup \rho(\hat{P}_{e''}) & \begin{array}[t]{l} e \in \{id, \texttt{new}\,c, \texttt{this}\}, \\ e = hd(e_m) \text{ where } e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp \text{ and } e'' \text{ is not a synch. exp.,} \\ e = hd(e_2) \text{ and } e_1 \text{ is not a synch. exp.} \\ \text{where } e_1\texttt{;}e_2,\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ \texttt{if}\ e_1\ \texttt{then}\ e_3\ \texttt{else}\ e_2, \\ \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)},\ e_1\texttt{.}m\texttt{(}e_2\texttt{)}, \text{ or } \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 \in \wp \end{array} \\
\{e_1\} & \begin{array}[t]{l} e \in \{id\ \texttt{:=}\ e_1, \texttt{throw}\ e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\} \\ \text{and } e_1 \text{ is a synch. exp.} \end{array} \\
\rho(\hat{P}_{e_1}) & \begin{array}[t]{l} e \in \{id\ \texttt{:=}\ e_1, \texttt{throw}\ e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\} \\ \text{and } e_1 \text{ is not a synch. exp.} \end{array} \\
\{e_2\} & e \in \{e_1\ \texttt{;}\ e_2, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2\} \text{ and } e_2 \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_2}) & e \in \{e_1\ \texttt{;}\ e_2, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2\} \text{ and } e_2 \text{ is not a synch. exp.} \\
\{e_1\} & \begin{array}[t]{l} e \in \{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)}\} \\ \text{and } e_1 \text{ is a synch. exp.} \end{array} \\
\rho(\hat{P}_{e_1}) & \begin{array}[t]{l} e \in \{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3, \texttt{try}\ e_1\ \texttt{catch (}c_1\ x_1\ e_2\texttt{)}\} \\ \text{and } e_1 \text{ is not a synch. exp.} \end{array} \\
\{e_m\} & e = e_1\texttt{.}m\texttt{(}e_2\texttt{)} \text{ and } e_m \text{ is a synch. exp.} \\
\rho(\hat{P}_{e_m}) & e = e_1\texttt{.}m\texttt{(}e_2\texttt{)} \text{ and } e_m \text{ is not a synch. exp.}
\end{cases}
$$

We prove $Q(\mathrm{fix}\mathcal{F})$ by the fixpoint induction, where the assertion $Q(\rho)$ for a program $\wp$ is:

$$\forall e \in \wp.\ \forall e' \in \rho(\hat{P}_e).\ \mathit{Time}(e') \leq \mathit{Time}(e) \quad \text{for any evaluation of } \wp.$$

Base case $Q(\emptyset)$ trivially holds. We prove that $Q(\mathcal{F}(\rho))$ holds given the induction hypothesis $Q(\rho)$. For brevity, we call synchronizing expressions *synch. exp.*

– Case $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e \neq hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$,
$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, $\forall e' \in \mathcal{F}(\rho)(\hat{P}_e). Time(e') \leq Time(e)$ holds.

– Case $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e \neq hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$,
$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– Case $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e = hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$ and e'' is a synch. exp.,
$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\texttt{.}m\texttt{(}e''\texttt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1, e''\}$, the case holds.

– Case $e \in \{id, \texttt{new}c, \texttt{this}\}$, $e = hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$ and e'' is not a synch. exp.,
$e = hd(e_2)$ and e_1 is a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\texttt{.}m\texttt{(}e''\texttt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e''})$. $Time(e') \leq Time(e'')$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\} \cup \rho(\hat{P}_{e''})$, the case holds.

– Case $e \in \{id, \texttt{new}\, c, \texttt{this}\}$, $e = hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$ and e'' is a synch. exp.,
$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\texttt{.}m\texttt{(}e''\texttt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1}) \cup \{e''\}$, the case holds.

– Case $e \in \{id, \texttt{new}c, \texttt{this}\}$, $e = hd(e_m)$ where $e'\texttt{.}m\texttt{(}e''\texttt{)} \in \wp$ and e'' is not a synch. exp.,
$e = hd(e_2)$ and e_1 is not a synch. exp. where $e_1\texttt{;}e_2 \in \wp$

Since $ExSq(e_1\texttt{;}e_2) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. And because $ExSq(e'\texttt{.}m\texttt{(}e''\texttt{)}) = ExSq(e') \oplus ExSq(e'') \oplus ExSq(e_m) \oplus \{e\}$, e'' is always evaluated earlier than $hd(e_m) = e$. Thus, $Time(e'') \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$ and $\forall e' \in \rho(\hat{P}_{e''})$. $Time(e') \leq Time(e'')$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1}) \cup \rho(\hat{P}_{e''})$, the case holds.

– Case $e \in \{id \texttt{ := } e_1, \texttt{throw}\, e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\}$ and e_1 is a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, the case holds.

– Case $e \in \{id \texttt{ := } e_1, \texttt{throw}\, e_1, e_1\texttt{.start}, e_1\texttt{.wait}, e_1\texttt{.notify}\}$ and e_1 is not a synch. exp.

Since $ExSq(e) = ExSq(e_1) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. By induction, $\forall e' \in \rho(\hat{P}_{e_1})$. $Time(e') \leq Time(e_1)$.

Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– `Case` $e \in \{e_1$ `;` $e_2,$`xthrow` e_1 `to` $e_2\}$ and e_2 is a synch. exp.
Since $ExSq(e) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_2)) = e_2$, e_2 is always evaluated earlier than e. Thus, $Time(e_2) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_2\}$, the case holds.

– `Case` $e \in \{e_1$ `;` $e_2,$`xthrow` e_1 `to` $e_2\}$ and e_2 is not a synch. exp.
Since $ExSq(e) = ExSq(e_1) \oplus ExSq(e_2) \oplus \{e\}$ and $last(ExSq(e_2)) = e_2$, e_2 is always evaluated earlier than e. Thus, $Time(e_2) \leq Time(e)$. By induction hypothesis, $\forall e' \in \rho(\hat{P}_{e_2}).\ Time(e') \leq Time(e_2)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_2})$, the case holds.

– `Case` $e \in \{$`if` e_1 `then` e_2 `else` $e_3,$`try` e_1 `catch (`$c_1\ x_1\ e_2$`)`$\}$ and e_1 is a synch. exp.
Since $ExSq($`if` e_1 `then` e_2 `else` $e_3) = ExSq(e_1) \oplus (ExSq(e_2) \cup ExSq(e_3)) \oplus \{e\}$, $ExSq($`try` e_1 `catch (`$c_1\ x_1\ e_2$`)`$) = (ExSq(e_1) \cup (ExSq(e_1) \oplus ExSq(e_2))) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_1\}$, the case holds.

– `Case` $e \in \{$`if` e_1 `then` e_2 `else` $e_3,$`try` e_1 `catch (`$c_1\ x_1\ e_2$`)`$\}$ and e_1 is not a synch. exp.
Since $ExSq($`if` e_1 `then` e_2 `else` $e_3) = ExSq(e_1) \oplus (ExSq(e_2) \cup ExSq(e_3)) \oplus \{e\}$, $ExSq($`try` e_1 `catch (`$c_1\ x_1\ e_2$`)`$) = (ExSq(e_1) \cup (ExSq(e_1) \oplus ExSq(e_2))) \oplus \{e\}$ and $last(ExSq(e_1)) = e_1$, e_1 is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. By induction, $\forall e' \in \rho(\hat{P}_{e_1}).\ Time(e') \leq Time(e_1)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \rho(\hat{P}_{e_1})$, the case holds.

– `Case` $e = e_1$`.`m`(`e_2`)` and e_m is a synch. exp.
Since $ExSq(e_1$`.`m`(`e_2`)`$) = ExSq(e_1) \oplus ExSq(e_2) \oplus ExSq(e_m) \oplus \{e\}$ and $last(ExSq(e_m)) = e_m$, e_m is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_m\}$, the case holds.

– `Case` $e = e_1$`.`m`(`e_2`)` and e_m is not a synch. exp.
Since $ExSq(e_1$`.`m`(`e_2`)`$) = ExSq(e_1) \oplus ExSq(e_2) \oplus ExSq(e_m) \oplus \{e\}$ and $last(ExSq(e_m)) = e_m$, e_m is always evaluated earlier than e. Thus, $Time(e_1) \leq Time(e)$. By induction, $\forall e' \in \rho(\hat{P}_{e_m}).\ Time(e') \leq Time(e_m)$. Because $\mathcal{F}(\rho)(\hat{P}_e) = \{e_m\}$, the case holds.

Other cases are similarly proved. □

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[DGC98, PS91, TP00]. 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

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

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

$$[\delta_3]\quad \frac{\begin{array}{c} e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \overset{c''\uparrow c'}{\not\longrightarrow} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \widehat{pre}(e_1.\texttt{notify}) \end{array}}{e.\texttt{wait} \xrightarrow{c\uparrow c'} hd(e_{c'.\texttt{run}})}$$

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

$$[\delta_5]\quad \frac{\begin{array}{c} e.\texttt{wait} \in c \quad e_1.\texttt{notify} \in c'' \quad c'' \neq c' \in \wp \quad e_2 \overset{c''\downarrow c'}{\not\longrightarrow} e_3 \\ \mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset \quad e_2 \in \widehat{suc}(e_1.\texttt{notify}) \end{array}}{e.\texttt{wait} \xrightarrow{c\downarrow c'} e_{c'.\texttt{run}}}$$

Figure 7: Concurrency Analysis Rules

enclosing method is called. This is formally defined as follows:

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

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

Figure 7 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 $[\delta_1]$:

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

Since e.`wait` in c may be notified by e_1.`notify` in c', these two expressions may be evaluated at the same time.

Consider rule $[\delta_2]$:

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

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 $[\delta_3]$:

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

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 $\mathit{hd}(e_{c'.\texttt{run}})$ because $\mathit{hd}(e_{c'.\texttt{run}})$ is the first expression evaluated in c'.

Rules in Figure 7 safely finds the expressions in c' which may be synchronized with the expressions in c before (after) the evaluation of e:

Lemma 5 *For a program $\wp$, let*

$$\delta_\uparrow(c, e, c') = \{e' \mid e \xrightarrow{c\uparrow c'} e' \text{ is deducible by the rules in Figure 7}\}$$
$$\delta_\downarrow(c, e, c') = \{e' \mid e \xrightarrow{c\downarrow c'} e' \text{ is deducible by the rules in Figure 7}\}.$$

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

Proof. Since the second property is a dual of the first one, we only prove the first property here.

1. We prove only the case of $\delta_\uparrow(c, e.\texttt{wait}, c')$ and the dual case of $\delta_\uparrow(c, e.\texttt{notify}, c')$ is similarly proved.

For any evaluation of $\wp$ and $e.\texttt{wait} \in c$ in $\wp$, there exists $e_1.\texttt{notify}$ such that $\mathit{Class}(e) \cap \mathit{Class}(e_1) \neq \emptyset$, which notifies $e.\texttt{wait} \in c$. Since $e_1.\texttt{notify}$ notifies $e.\texttt{wait}$, $\mathit{Time}(e_1.\texttt{notify}) = \mathit{Time}(e.\texttt{wait})$.

We prove for two cases of $e_1.\texttt{notify}$: (1) $e_1.\texttt{notify} \in c'$ and (2) $e_1.\texttt{notify} \in c''$ $c' \neq c'' \in \wp$.

(1) $e_1.\texttt{notify} \in c'$

By $[\delta_1]$, $e.\texttt{wait} \xrightarrow{c\uparrow c'} e_1.\texttt{notify}$. Therefore, $e_1.\texttt{notify} \in \delta_\uparrow(c, e.\texttt{wait}, c')$ and $\mathit{Time}(e_1.\texttt{notify}) = \mathit{Time}(e.\texttt{wait})$.

(2) e_1`.notify` $\in c''$ $c' \neq c'' \in \wp$

We need to show that there exists $e' \in \delta_\uparrow(c, e\texttt{.wait}, c')$ s.t. $Time(e') \leq Time(e\texttt{.wait})$ or $e' = hd(e_{c'}\texttt{.run})$. And it is equivalent to show that $e\texttt{.wait} \xrightarrow{c\uparrow c'} e'$ such that $Time(e') \leq Time(e\texttt{.wait})$ or $e' = hd(e_{c'}\texttt{.run})$ is deducible. We prove that, for any size of derivation tree, there exists $e\texttt{.wait} \xrightarrow{c\uparrow c'} e'$ such that $Time(e') \leq Time(e\texttt{.wait})$ or $e' = hd(e_{c'}\texttt{.run})$.

– `Case` leaves: By $[\delta_3]$, $e = hd(e_{c'}\texttt{.run})$.

– `Case` internal nodes: By $[\delta_2]$, there exist $e_2 \in \hat{pre}(e_1\texttt{.notify})$ and $e_2 \xrightarrow{c''\uparrow c'} e_3$. By Lemma 4, $\forall e_2 \in \hat{pre}(e_1\texttt{.notify})$. $Time(e_2) \leq Time(e_1\texttt{.notify})$. By induction, $Time(e_3) \leq Time(e_2)$ or $e_3 = hd(e_{c'}\texttt{.run})$. If $Time(e_3) \leq Time(e_2)$, then $Time(e_3) \leq Time(e_2) \leq Time(e_1\texttt{.notify}) \leq Time(e\texttt{.wait})$. □

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

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

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 prove 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. For a program $\wp$ and any evaluation of the program, let e' be evaluated in c' concurrently with e in c, then

$$Time(e) = Time(e'). \tag{1}$$

We need to show that e' is included in $Concur(c, e, c')$. By Lemma 4,

$$\forall e_1 \in \hat{pre}(e). Time(e_1) \leq Time(e) \tag{2}$$

and

$$\forall e_2 \in s\hat{u}c(e).\, Time(e_2) \geq Time(e). \tag{3}$$

By Lemma 5, there exists $e'_1 \in \delta_{\uparrow}(c, e_1, c')$ such that

$$Time(e'_1) \leq Time(e_1) \tag{4}$$

or

$$e'_1 = hd(e_{c'.\mathtt{run}}) \tag{5}$$

and there exists $e'_2 \in \delta_{\downarrow}(c, e_2, c')$ such that

$$Time(e'_2) \geq Time(e_2) \tag{6}$$

or

$$e'_2 = e_{c'.\mathtt{run}}. \tag{7}$$

For the case of (4), we get

$$Time(e'_1) \leq Time(e')$$

from (1), (2) and (4). For the other case of (5), we also get

$$Time(e'_1) \leq Time(e')$$

since $hd(e_{c'.\mathtt{run}})$ is the first expression evaluated in c'. Similarly, for the case of (6), we get

$$Time(e'_2) \geq Time(e')$$

from (1), (3) and (6). And for the other case of (7), we also get

$$Time(e'_2) \geq Time(e')$$

since $e_{c'.\mathtt{run}}$ is the last expression evaluated in c'. Thus

$$Time(e'_1) \leq Time(e') \leq Time(e'_2),$$

and then $e' \in suc^*(e'_1) \cap pre^*(e'_2)$ by Lemma 1. Hence, e' is included in $Concur(c, e, c')$. Therefore, every expression which may be evaluated in c' concurrently with e in c is included in $Concur(c, e, c')$. $\square$

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 [Hei92, AH95]. As we mentioned before, we assume a safe class information $\mathit{Class}(e)$ for every expression e and we use the safe concurrency information $\mathit{Concur}(c, e, c')$ (Section 3) for every pair of $e \in c$ and c' in the exception analysis.

We present two set-constraint systems and our analysis is the second one. Section 4.1 presents a constraint system that estimates uncaught exceptions from every expression of the input program. Because exception-related expressions are sparse in programs, it is wasteful to generate constraints for every expression of the program. Thus, in Section 4.2, we present a sparse constraint system that increases the constraint granularity to the method-level. This constraint system analyzes uncaught exceptions from every method of the input program. Similar techniques of designing a sparse constraint system that analyzes the properties at a larger granularity than at every expression have already been found to be cost-effective in ML [MTHM97]'s exception analysis [YR97, YR01].

4.1 Exception Analysis at Expression-Level

Figure 8 shows our rules to generate set constraints for the uncaught exception classes of every expression. Every expression e of the input program has a set constraint: $X_e \supseteq se$. The set variable X_e is for the exception classes that the uncaught exceptions during e's evaluation belong to. A constraint $X_e \supseteq se$ may be read as "an expression e has uncaught exceptions including those of se" and a relation $\triangleright_1 e : \mathcal{C}$ may be read as "a set of constraints $\mathcal{C}$ is generated from an expression e."

Consider the rule for `throw` expression:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1}{\triangleright_1 \texttt{throw}\ e_1 : \{X_e \supseteq \mathit{Class}(e_1) \cup X_{e_1}\} \cup \mathcal{C}_1}.$$

It may have uncaught exceptions from inside e_1 (X_{e_1}) or throws exceptions e_1 ($\mathit{Class}(e_1)$).

Consider the rule for `try` expression:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1 \quad \triangleright_1 e_2 : \mathcal{C}_2}{\triangleright_1 \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \{X_e \supseteq (X_{e_1} - \{c_1\}^*) \cup X_{e_2}\} \cup\ \mathcal{C}_1 \cup \mathcal{C}_2}.$$

If the classes of raised exceptions from e_1 are covered by c_1, the raised exceptions are caught by x_1 ($X_{e_1} - \{c_1\}^*$, where $\{c\}^*$ represents all the subclasses of a class c). While handling the uncaught exceptions from e_1 in e_2, this `try` expression may have uncaught exceptions from inside e_2 (X_{e_2}).

In the rule for method call:

$$\frac{\triangleright_1 e_1 : \mathcal{C}_1 \quad \triangleright_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\triangleright_1 e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \{X_e \supseteq X_{c'.m} \mid c' \in \mathit{Class}(e_1), m(x) = e_m \in c'\} \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \\ \qquad\qquad\qquad \cup\ \mathcal{C}_1 \cup \mathcal{C}_2\end{array}},$$

it may have uncaught exceptions from the subexpressions e_1 and e_2 ($X_{e_1} \cup X_{e_2}$), or from the body expression e_m of the method m which is defined inside the classes

[Program]$_1$ $$\frac{\rhd_1 C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_1 C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[ClassDef]$_1$ $$\frac{\rhd_1 M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_1 \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[MethDef]$_1$ $$\frac{\rhd_1 e_m : \mathcal{C}}{\rhd_1 m(x)\ \texttt{=}\ e_m : \{X_{c.m} \supseteq X_{e_m}\} \cup \mathcal{C}}$$

[Variable]$_1$ $$\rhd_1 id : \emptyset$$

[Assign]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 id\ \texttt{:=}\ e_1 : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

[New]$_1$ $$\rhd_1 \texttt{new}\ c : \emptyset$$

[This]$_1$ $$\rhd_1 \texttt{this} : \emptyset$$

[Seq]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\rhd_1 e_1\ \texttt{;}\ e_2 : \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2 \quad \rhd_1 e_3 : \mathcal{C}_3}{\rhd_1 \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \{X_e \supseteq X_{e_1} \cup X_{e_2} \cup X_{e_3}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

[Throw]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 \texttt{throw}\ e_1 : \{X_e \supseteq \mathit{Class}(e_1) \cup X_{e_1}\} \cup \mathcal{C}_1}$$

[Try]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\rhd_1 \texttt{try}\ e_1\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_2\texttt{)} : \{X_e \supseteq (X_{e_1} - \{c_1\}^*) \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[MethCall]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{c}\rhd_1 e_1\texttt{.}m\texttt{(}e_2\texttt{)} : \{X_e \supseteq X_{c'.m} \mid c' \in \mathit{Class}(e_1), m(x) = e_m \in c'\} \\ \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

[ThdStart]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.start} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

[Wait]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.wait} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

[Notify]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1}{\rhd_1 e_1\texttt{.notify} : \{X_e \supseteq X_{e_1}\} \cup \mathcal{C}_1}$$

[Xthrow]$_1$ $$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\rhd_1 \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{e'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ \qquad\qquad c' \in \mathit{Class}(e_2), \\ \qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \quad \cup \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}$$

Figure 8: Exception Analysis at Expression-Level

$c' \in \mathit{Class}(e_1)$ of e_1's objects ($X_e \supseteq X_{c'.m}$, where the subscript $c'.m$ indicates the body expression of class c''s method m).

Finally, consider the rule for `xthrow` expression:

$$\frac{\rhd_1 e_1 : \mathcal{C}_1 \quad \rhd_1 e_2 : \mathcal{C}_2}{\begin{array}{l}\rhd_1 \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{e'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2),\ c' \in \mathit{Class}(e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \qquad\qquad\qquad \cup\ \{X_e \supseteq X_{e_1} \cup X_{e_2}\} \cup \mathcal{C}_1 \cup \mathcal{C}_2\end{array}}.$$

Uncaught exceptions from `xthrow` expression are those from the subexpressions e_1 and e_2 ($X_{e_1} \cup X_{e_2}$). Note that exceptions e_1 are thrown to the expressions e' ($X_{e'} \supseteq \mathit{Class}(e_1)$) which may be evaluated in the thread $c' \in \mathit{Class}(e_2)$ concurrently with the evaluation of `xthrow` expression in the thread c ($\mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')$).

Our exception analysis at expression-level is safe:

Theorem 2 *For a program $\wp$, let $\rhd_1 \wp : \mathcal{C}$. Then for every sub-expression e of $\wp$, $lm(\mathcal{C})(X_e)$ includes all the exceptions that escape from e during the execution of $\wp$.*

Proof Sketch. This correctness can be proved by the fixpoint induction over the continuous functions that are derived [CC95] from our constraint system [YR01]. Note that the correctness of our exception analysis relies on the correctness of both the class analysis and the concurrency analysis. □

4.2 Exception Analysis at Method-Level

Our new sparse constraint system is presented in Figure 9. Since this system generates constraints only for class' methods and `try`-blocks, the number of set variables becomes proportional to the number of methods and `try`-blocks instead of the number of expressions. For each method m and `try`-block e_g in "`try` e_g `catch` $(c\ x\ e)$", 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 $m \rhd_2 e : \mathcal{C}$ ($g \rhd_2 e : \mathcal{C}$) 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:

$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ \texttt{throw}\ e_1 : \{X_m \supseteq \mathit{Class}(e_1)\} \cup \mathcal{C}_1}.$$

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

Consider the rule for `try` expression:

$$\frac{g \rhd_2 e_g : \mathcal{C}_g \quad m \rhd_2 e_1 : \mathcal{C}_1}{m \rhd_2 \texttt{try}\ e_g\ \texttt{catch}\ (c_1\ x_1\ e_1) : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup\ \mathcal{C}_g \cup \mathcal{C}_1}.$$

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.

[Program]$_2$
$$\frac{\rhd_2 C_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_2 C_1, \cdots, C_n : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[ClassDef]$_2$
$$\frac{\rhd_2 M_i : \mathcal{C}_i,\ i = 1, \cdots, n}{\rhd_2 \texttt{class}\ c\ \texttt{ext}\ c'\ \{\texttt{var}\ x^*\ M^*\} : \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_n}$$

[MethDef]$_2$
$$\frac{m \rhd_2 e_m : \mathcal{C}}{\rhd_2 m\texttt{(}x\texttt{)}\ \texttt{=}\ e_m : \mathcal{C}}$$

[Variable]$_2$
$$m \rhd_2 id : \emptyset$$

[Assign]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ id\ \texttt{:=}\ e_1 : \mathcal{C}_1}$$

[New]$_2$
$$m \rhd_2 \texttt{new}\ c : \emptyset$$

[This]$_2$
$$m \rhd_2 \texttt{this} : \emptyset$$

[Seq]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1\ \ m \rhd_2 e_2 : \mathcal{C}_2}{m \rhd_2\ e_1\ \texttt{;}\ e_2 : \mathcal{C}_1 \cup \mathcal{C}_2}$$

[Cond]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\ \ m \rhd_2 e_2 : \mathcal{C}_2\ \ m \rhd_2 e_3 : \mathcal{C}_3}{m \rhd_2 \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3}$$

[Throw]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ \texttt{throw}\ e_1 : \{X_m \supseteq \mathit{Class}(e_1)\} \cup \mathcal{C}_1}$$

[Try]$_2$
$$\frac{g \rhd_2 e_g : \mathcal{C}_g\ \ m \rhd_2 e_1 : \mathcal{C}_1}{m \rhd_2 \texttt{try}\ e_g\ \texttt{catch}\ \texttt{(}c_1\ x_1\ e_1\texttt{)} : \{X_m \supseteq (X_g - \{c_1\}^*)\} \cup \mathcal{C}_g \cup \mathcal{C}_1}$$

[MethCall]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\ \ m \rhd_2 e_2 : \mathcal{C}_2}{m \rhd_2\ e_1\texttt{.}m'\texttt{(}e_2\texttt{)} : \{X_m \supseteq X_{c'.m'} \mid c' \in \mathit{Class}(e_1), m'(x) = e_{m'} \in c'\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}$$

[ThdStart]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.start} : \mathcal{C}_1}$$

[Wait]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.wait} : \mathcal{C}_1}$$

[Notify]$_2$
$$\frac{m \rhd_2\ e_1 : \mathcal{C}_1}{m \rhd_2\ e_1\texttt{.notify} : \mathcal{C}_1}$$

[Xthrow]$_2$
$$\frac{m \rhd_2 e_1 : \mathcal{C}_1\ \ m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd_2\ \texttt{xthrow}\ e_1\ \texttt{to}\ e_2 : \{X_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2), \\ \qquad c' \in \mathit{Class}(e_2), e' \in e_{m'}, \\ \qquad m'(x) = e_{m'} \in \wp, \\ \qquad e' \in \mathit{Concur}(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')\} \\ \cup\ \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}$$

Figure 9: Exception Analysis at Method-Level

Finally, consider the rule for `xthrow` expression:

$$\frac{m \rhd_2 e_1 : \mathcal{C}_1 \quad m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m \;\; \rhd_2 \;\; \texttt{xthrow}\, e_1 \;\texttt{to}\; e_2 : \{X_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\, e_1 \;\texttt{to}\; e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad c' \in \mathit{Class}(e_2), m'(x) = e_{m'} \in \wp, e' \in e_{m'}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad e' \in \mathit{Concur}(c, \texttt{xthrow}\, e_1 \;\texttt{to}\; e_2, c')\} \\ \qquad\qquad\qquad\quad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}.$$

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

The method-level exception constraint rule $\rhd_2$ is a safe approximation of $\rhd_1$:

Theorem 3 *For a program $\wp$, let $\rhd_1 \wp : \mathcal{C}_1$ and $\rhd_2 \wp : \mathcal{C}_2$. Then, if $m \;\; \rhd_2 \;\; e : \; \mathcal{C}$ occurs during $\rhd_2 \; \wp \; : \; \mathcal{C}_2$ then $\mathit{lm}(\mathcal{C}_2)(X_m) \supseteq \mathit{lm}(\mathcal{C}_1)(X_e)$.*

Proof. This correctness proof is an instance of the correctness proof in [YR01]. □

4.3 Separate Constraints for Improved Accuracy

We can safely improve the accuracy of our exception analysis. Since the thrown exceptions inside a thread do not propagate across the thread, it is possible to localize the effect of throwing asynchronous exceptions: for each method m, uncaught exceptions thrown from other threads are collected in X^T_m rather than in X_m. The rule for `xthrow` expression becomes:

$$\frac{m \rhd_2 e_1 : \mathcal{C}_1 \quad m \rhd_2 e_2 : \mathcal{C}_2}{\begin{array}{l} m \rhd_2 \texttt{xthrow}\, e_1 \;\texttt{to}\; e_2 : \{X^T_{m'} \supseteq \mathit{Class}(e_1) \mid c \in \mathit{owner}(\texttt{xthrow}\, e_1 \;\texttt{to}\; e_2), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad c' \in \mathit{Class}(e_2), m'(x) = e_{m'} \in \wp, e' \in e_{m'}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad e' \in \mathit{Concur}(c, \texttt{xthrow}\, e_1 \;\texttt{to}\; e_2, c')\} \\ \qquad\qquad\qquad\quad \cup\, \mathcal{C}_1 \cup \mathcal{C}_2 \end{array}}.$$

After constructing constraints for a given input program, only the set constraint $X_m \supseteq se$ for each method m is used during the first resolution phase. At the second phase, the set constraint $X^T_m \supseteq se$ for m is added to the partially solved system and then solved completely. The correctness of this improved $\rhd_2$ can be proved with respect to an improved $\rhd_1$.

5 Analysis Complexity

In this section, we analyze the complexity of our method. Since our analysis is divided into several analyses, the complexity of our analysis is a sum of the complexities of all sub-analyses.

The complexity of the time analysis among expressions *pre* and *suc* is linear to program size n. Constructing set constraints is linear to n and solving constraints is not need here because set expressions are just unions of expressions.

And the complexity of the time analysis among synchronizing expressions $\widehat{pre}$ and $\widehat{suc}$ is $O(n^3)$. Constructing constraints is linear to n and computing the constraints' least model uses the conventional fixpoint iteration of cubic complexity. The iterative method is possible because the set domain is finite (the set of expressions in the input program) and the set operator $\cup$ is monotonic.

The complexity of the concurrency analysis *Concur* is $O(n^5)$. First, the complexity of $\delta_\uparrow$ and $\delta_\downarrow$ is $O(n^5)$. Because for each c and c' in the program ($O(n^2)$ combinations), computing $\delta_\uparrow(c, e, c')$ and $\delta_\downarrow(c, e, c')$ uses the fixpoint iteration of cubic complexity. Second, the complexity of pre^* and suc^* is $O(n^3)$. Since pre^* (suc^*) applies pre (suc) at most n times and, for each application of pre (suc), there is at most n arguments of pre (suc) and n expressions may be added. At last, for each $e_1 \in \hat{pre}(e)$, $e_2 \in \hat{suc}(e)$, $e'_1 \in \delta_\uparrow(c, e_1, c')$, and $e'_2 \in \delta_\downarrow(c, e_2, c')$ ($O(n^4)$ combinations), *Concur* collects expressions in $suc^*(e'_1) \cap pre^{(}e'_2)$ ($O(n)$).

Finally, the complexity of our exception analysis at method-level is ($O(n^3)$). Since our construction of set constraints is not compositional, constructing constraints is not linear to n. Consider rule [Xthrow]$_2$. For each $c \in owner(\texttt{xthrow}\ e_1\ \texttt{to}\ e_2)$ and $c' \in Class(e_2)$ ($O(n^2)$ combinations), it collects $e' \in Concur(c, \texttt{xthrow}\ e_1\ \texttt{to}\ e_2, c')$ ($O(n)$). And rule [MethCall] collects $c'.m'$ for each $c' \in Class(e_1)$ ($O(n)$). Other rules generates constraints linear to program size. Thus, constructing constraints is $O(n)$ and computing the constraints' least solution uses the fixpoint iteration of cubic complexity because the set domain is finite and the set operators $\cup$ and $X_g - \{c_1\}^*$ are monotonic.

6 Related Work

Several exception analyses have been developed to detect uncaught exceptions in ML programs [Yi94, YR97, YR01, LP00]. Yi first designed an exception analysis for Standard ML (SML) programs based on abstract interpretation [Yi94], which was very accurate but too slow. Yi and Ryu redesigned the analysis by set-based framework [YR97, YR01], which shows a good cost-accuracy performance. Fähndrich and Aiken [FFAC98] 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 [LP00] 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 [YC99] 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 [RM99] 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 multithreaded exception handling, several work were proposed to deal with the problem. Lea [Lea99] 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 [IH98] 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.

7 Conclusion

We have presented an exception analysis for 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 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

[AH95] Alex Aiken and Nevin Heintze. Constraint-based program analysis. Tutorial Notes of the ACM Symposium on Principles of Programming Languages, January 1995.

[CC95] Patrick Cousot and Radhia Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 170–181, 1995.

[CKRW97] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. From sequential to multi-threaded java: An event-based operational semantics. In Michael 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.

[CKRW99] Pietro Cenciarelli, Alexander Knapp, Bernhard Reus, and Martin Wirsing. An event-based structural operational semantics of multi-threaded java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer–Verlag, 1999.

[Dep] 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`.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[FFAC98] Manuel Fähndrich, Jeffrey S. Foster, Alexander Aiken, and Jason Cu. Tracking down exceptions in standard ml programs. Technical Report UCB/CSD-98-996, Computer Science Division, University of California, Berkeley, February 1998.

[GJJB00] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java^{TM} Language Specification.* Addison Wesley, second edition edition, 2000.

[Hei92] Nevin Heintze. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, October 1992.

[IH98] Joe De Russo III and Peter Haggar. Multithreaded exception handling in java. Java Report, August 1998.

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

[Lea99] Doug Lea. *Concurrent Programming in Java^{TM}, Second Edition: Design Principles and Patterns.* Addison-Wesley, 1999.

[LP00] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, March 2000.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[PS91] Jens Palsberg and Michael I. Schwarzbach. Object-oriented type inference. In *Proceedings of ACM Conference on OOPSLA*, pages 141–161, 1991.

[RM99] Martin P. Robillard and Gail 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.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of ACM Conference on OOPSLA*, pages 281–293, 2000.

[YC99] Kwangkeun Yi and Byeong-Mo 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.

[Yi94] Kwangkeun 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.

[YR97] Kwangkeun Yi and Sukyoung 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.

[YR01] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 273(1), 2001. Extended version of [YR97] (to appear).