1-15-2013

On Exceptions, Events and Observer Chains

Mehdi Bagherzadeh  
Iowa State University, mbagherz@iastate.edu

Hridesh Rajan  
Iowa State University

Ali Darvish  
Iowa State University, ali2@iastate.edu

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons

Recommended Citation
Bagherzadeh, Mehdi; Rajan, Hridesh; and Darvish, Ali, "On Exceptions, Events and Observer Chains" (2013). Computer Science Technical Reports. 300.
http://lib.dr.iastate.edu/cs_techreports/300

This Article is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Keywords: event, exceptional behavior, exception flow

Categories and Subject Descriptors:
D.3.1 [Programming Languages]: Formal Definitions and Theory—Semantics
D.3.3 [Programming Languages]: Language Constructs and Features—Exceptions
D.2.4 [Software/Program Verification]: Programming by contract, Assertion checkers

Copyright (c) 2012, Mehdi Bagherzadeh, Hridesh Rajan and Ali Darvish.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA
ABSTRACT

Modular understanding of behaviors and flows of exceptions may help in their better use and handling. Such reasoning tasks about exceptions face unique challenges in event-based implicit invocation (II) languages that allow subjects to implicitly invoke observers, and run the observers in a chain. In this work, we illustrate these challenges in Ptolemy and propose Ptolemy’s exception-aware specification expressions and boundary exceptions limit the set of (un)checked exceptions of subjects and observers of an event. Exceptional postconditions specify the behaviors of these exceptions. Greobox specifications specify the flows of these exceptions among the observers in the chain. Ptolemy’s type system and refinement rules enforce these specifications and thus enable its modular reasoning. We evaluate the utility of Ptolemy’s exception flow reasoning by applying it to understand a set of aspect-oriented (AO) bug patterns. We also present Ptolemy’s semantics including its sound static semantics.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Exceptions

General Terms Languages, Theory

Keywords event, exceptional behavior, exception flow

1. INTRODUCTION

Exceptions are useful for structured separation of normal and error handling code. However, their improper use and handling could put a system in undetermined risky states or even crash it [30]. Understanding exceptions, especially their behaviors and flows may help with their better use and handling [29, 35]. Previous work, such as JML [25], ESC/Java [12], the work of Jacobs et al. [16] anchored exceptions [40], Jex [54] and EJFlow [4] enable reasoning about behaviors or flows of exceptions. However, they are tailored for systems in which invocation relations among the modules are explicit and known, i.e. explicit invocation (EI) [7]. With EI, in languages such as Java, a module explicitly invokes another module with a method call $E.m()$ in which both the static type of the invoked module $E$ and the name of the method $m$ are known. EI reasoning techniques use this knowledge to incorporate the behaviors and flows of the exceptions of the method $m$ into its invoking module [7]. Supertype abstraction enables reasoning independent of the dynamic type of $E$. However, the invocation relations among modules of a system are not limited to explicit invocations. This is true in languages such as Java or C# when using events and delegates or in languages such as AspectJ [21,22] or Ptolemy [32] when a module invokes another module implicitly and without knowing about it, i.e. implicit invocation (II) [15]. Ptolemy is an event-based II extension of Java.

Modular reasoning about behaviors and flows of exceptions faces unique challenges in II languages such as Ptolemy or AspectJ that allow a (subject) module to invoke other (observer) modules without knowing about them and run them in a chain. In Ptolemy, a subject announces an event and zero or more observers register for the event and handle it. The observers form a chain based on their dynamic registration order and may invoke each other. The observers are not explicitly mentioned in the subject and are invoked implicitly by an II mechanism, upon announcement of the event. The following code snippet in Ptolemy illustrates a subject module that announces an event $Ev$ causing its unknown observers to be invoked in a chain.

```
announce $Ev($)
```

To reason about this subject, especially behaviors and flows of exceptions during announcement and handling of $Ev$, behaviors of its observers when throwing the exceptions and flows of these exceptions in the chain of the observers should be understood. Depending on the order of the observers in the chain, an exception thrown by one observer may be caught by another observer or propagated down the chain. However, the observers, their orders in the chain, and the behaviors and flows of their exceptions are not known to the subject. Even if the such information were available for individual observers, a naive use of EI reasoning techniques may require all possible orders of execution of the observers to be considered, i.e. $n!$ for $n$ observers. Such dependency of reasoning on system configuration, i.e. individual observers, their order and behaviors and flows of their exceptions, threatens its modularity since any changes in the system configuration could invalidate any previous reasoning. The main difference from Java’s Event Model or the Observer pattern [24] is that observers form a chain based on their dynamic registration order and may invoke each other.

Previous work on modular reasoning about Ptolemy-like II languages, such as translucid contracts [1], join point interfaces (JPI) [27], crosscutting programming interfaces (XPI) and the work of Khatchadourian et al. [20], provides the subject with the knowl-
2. PROBLEM

In this section we illustrate the problems (1)–(3) for modular reasoning about (i) behaviors and (ii) flows of exceptions during announcement and handling of events in Ptolemy.

2.1 Modular Reasoning about Behaviors of Exceptions

As an example of modular reasoning about behaviors of exceptions during event announcement and handling, consider verification of the JML, like assertion $\Phi_0$ on line 11 of Figure 1. Figure 1 and Figure 2 illustrate a savings bank account with a withdraw functionality. The assertion $\Phi_0$ says that: an account is not withdrawn if an exception $RegDExc$ is thrown during the announcement and handling of the event $WithdrawEv$, lines 5–8.

In other words the balance of the account is not changed if the announcement and handling of WithdrawEv terminates abnormally by throwing a $RegDExc$ exception. The expression old refers to the values of variables at the beginning of the method withdraw.

To verify the assertion $\Phi_0$, one should understand the behavioral of the announcement and handling of the event $WithdrawEv$, lines 5–8, for the exception $RegDExc$. This involves understanding the exceptional behaviors of the unknown observers of $WithdrawEv$, which are invoked in a chain upon its announcement, and also the exceptional behavior of the body of the announce expression itself, lines 6–7. The exceptional behavior of an observer for an exception is the state of the observer right before throwing the exception.

For such understanding to be modular, one may only use the implementation and the interface of the subject module $Savings$, lines 1–14, and the interfaces it references, i.e., event type $WithdrawEv$, lines 20–28 of Figure 2.

The reasoning must be independent of system configuration, i.e., unknown observers of $WithdrawEv$ or their execution order in their chain. However, neither the implementation of the subject nor the event type declaration provides any knowledge about the exceptions that the observers of $WithdrawEv$ may throw, i.e., problem (4), or their exceptional behaviors if they terminate abnormally by throwing $RegDExc$, i.e. problem (2). To better understand these problems we provide a short background on Ptolemy.

2.1.1 Ptolemy Language in a Nutshell

The language Ptolemy is an extension of Java with support for explicit announcement and handling of typed events. In Figure 1 written in Ptolemy, the subject module $Savings$ explicitly announces the event $WithdrawEv$ using the announce expression, lines 5–8 with the event body on lines 6–7. The observer module $Check$, lines 15–27 shows interest in the event using the when – do binding declaration, line 24, and dynamically registers itself to handle it using the register expression, line 25. The observer $Check$ runs the observer handler method $check$ upon announcement of $WithdrawEv$. The observer $Check$ checks for undesired behaviors of withdraw, such as overdrawing or violation of maximum number of withdrawals of accounts, etc. For brevity, $Check$ only checks for maximum number of monthly withdrawals of savings accounts that is limited to 6 according to U.S. Federal Reserve board Regulation D. The field $numWithdrawals$ keeps track of the number of withdrawals. The observer $Check$ throws an exception $RegDExc$ if Regulation D is violated, line 20.

In Ptolemy, the subject $Savings$ and the observer $Check$ know about the event $WithdrawEv$, however, they do not know about each other which in turn means that the subject does not know about its observers or their exceptions, i.e., problem (1).

In Ptolemy, zero or more observers could register for the same event. Similar to AspectJ, these observers form a chain.
class Savings {
    int bal; int numWithdrawals;
    void withdraw(int amt) throws Throwable {
        try {
            announce WithdrawEv(this, amt);
            bal -= amt;
        } catch (NullPointerException e) {
            throw new RegDExc();
        } catch (RegDExc e) {
            // assert this.bal==old(this.bal);
        } catch (RuntimeException e) {
            e.printStackTrace();
        }
        next.invoke();
        nextWithdrawals++;
    }
    class Check {
        void check(WithdrawEv next) throws Throwable {
            refining
            establishes next.acc().bal==old(next.acc()).bal;
            if (next.acc().numWithdrawals>=6)
            throw new RegDExc();
        }
    }
classes throws Throwable in the signature of the observer handler methods do  
not address the problem. Previous work on join point interfaces (JPI) [2] solves this problem for checked exceptions by specifying the set of checked exceptions that the observers may throw. Following JPI, we limit the set of checked exceptions of the observers of an event and call these exceptions boundary exceptions. Figure 3 illustrates the boundary exception RegDExc for the observers of WithdrawEv in its declaration, line 29. The boundary exception RegDExc also limits the set of the checked exceptions of the subject Savings. With the boundary exception RegDExc, the throws clauses of the methods withdraw and check should be changed to throws RegDExc since it is the only checked exception they can throw.

```java
29 void event WithdrawEv throws RegDExc {
30     ... /* the same as before */
31 }
```

Figure 3: boundary exception RegDExc, line 29

However, the boundary exception of WithdrawEv does not say anything about the behaviors and flows of RegDExc or the unchecked exceptions of its observers, their behaviors or flows in the chain of the observers, i.e. problems (2)-(3).

2.2 Modular Reasoning about Flows of Exceptions

Flows of Checked Exceptions As an example of modular reasoning about flows of checked exceptions during event announcement and handling, consider verification of a requirement \( \Phi_{f1} \) that says: the checked boundary exception RegDExc must be propagated back to the subject Savings, if thrown by any observer of WithdrawEv during its announcement and handling. In other words, an exception RegDExc thrown by one observer of WithdrawEv must not be caught by another one of its observers in the chain of the observers. The checked exception RegDExc is used by the observer Check to tell the subject that the withdrawal operation terminated unsuccessfully. The subject in turn propagates the exception back to its clients, line 12, which supposedly have more contextual information to handle it. If the exception RegDExc is thrown but does not reach the subject, then the subject and consequently its clients may wrongly assume the successful termination of the withdraw, which is an undesired behavior.

```java
1 class Logger {
2     void log(WithdrawEv next) throws RegDExc {
3         try {
4             establishing next.acc().bal==old(next.acc()).bal{
5                 Log.logTrans(next.acc(),"withdraw");
6             }
7             next.invoke();
8         } catch (Exception e) { /* swallow */
9         }
10         when WithdrawEv do log;
11     }
12 }
```

Figure 4: Observer Logger could swallow RegDExc depending on its execution order in the chain of the observers.

This could happen if another observer, say Logger 1 in Figure 4, registers and runs before Check in the chain. We are using Translucid contracts in Ptolemy [1] for client contracts of the observers. The Observer Logger registers and handles of exceptions and only work for normal execution. One way of treating try-catch expressions in observers such as Logger servers. Now when WithdrawEv is announced, the observer Logger runs and invokes the observer Check using its invoke expression, line 8, which throws a RegDExc. However, the try – catch expression around the invoke expression of Logger, lines 9 and 10, catches the exception and swallows it, line 10.

To reason about and verify \( \Phi_{f1} \), one should understand the flow of RegDExc among the observers of WithdrawEv in the chain of the observers. For the reasoning to be modular, one may only use the implementation and interface of the subject Savings and the declaration of the event WithdrawEv, independent of system configuration. However, neither the subject nor the event provides any knowledge about the flows of the exceptions among the observers, i.e. problem (3).

The execution order of the observers in their chain is also important in reasoning about \( \Phi_{f1} \). If Logger runs before Check then \( \Phi_{f1} \) is violated because Logger swallows RegDExc thrown by Check. However, if the execution order is reversed, i.e. Check runs before Logger, then RegDExc propagates back to the subject and \( \Phi_{f1} \) holds.

Flows of Unchecked Exceptions The boundary exception of WithdrawEv tells the subject Savings that its observers may only throw the checked exception RegDExc, however, it does not say anything about their unchecked exceptions. To handle the unknown unchecked exceptions of the observers the subject Savings catches these exceptions using catch (RuntimeException e) {...} and prints out their backtraces, line 13. However, this catch clause could be unused especially if no unchecked exceptions of the observers reaches the subject. We refer to this property as \( \Phi_{f2} \). Such an unused catch clause, which is trying to catch exceptions that do not reach it, is an example of a bad programming practice. Coelho et al. [6] categorizes such unused catch clauses as residual catch bugs. Bug patterns are discussed in more details in Section 5

Again, different orders of the execution of the observers in their chain entails different results regarding \( \Phi_{f2} \). The catch clause on line 13 of Figure 1 is unused, if the observer Logger runs before other observers of WithdrawEv in the chain. Recall that in Ptolemy the body of the announce expression is at the end of the chain. The try-catch expression around the invoke expression in Logger, lines 9 and 10, catches and swallows not only the checked boundary exception RegDExc but also all unchecked exceptions of the observers which run after it in the chain, before these exceptions reach the subject Savings. However, if an observers runs before Logger in the chain and throws an unchecked exception then this exception could reach the subject Savings and thus its catch clause will actually be used and necessary.

3. Ptolemy

In this section we describe Ptolemy’s core syntax and specification features for stating behaviors and flows of exceptions during event announcement and handling and address the problems (1)-(4).

3.1 Program Syntax

Ptolemy, is an exception-aware extension of Ptolemy [52] with support for specification and modular reasoning about behaviors and flows of exceptions during event announcement and handling. Similar to Ptolemy, Ptolemy is an implicit invocation (II) object-oriented (OO) language with support for explicit announcement and handling ofyped events. The formal definition of Ptolemy is given as an expression language:

```plaintext`
```
The syntax of Ptolemy₃’s executable programs is shown in Figure 5. The syntax supports (typed) events, exceptions, classes, objects, inheritance and subtyping for classes. Similar to Java, exceptions are treated like objects and are divided into checked and unchecked exceptions. For simplicity, Ptolemy₃ does not have packages, privacy modifiers, abstract classes and methods, static members, interfaces or constructors. The superscript term shows a sequence of zero or more term whereas [term] means zero or one, i.e. optional. A Ptolemy₃ program (prog) is a collection of declarations (decl) followed by an expression (e), which is like a main method in Java. There are two types of declarations in Ptolemy₃; class and event type declarations.

\[
\begin{align*}
\text{prog} &::= \text{decl} \; \text{e} \\
\text{decl} &::= \text{class} \; \text{extends} \; \text{d} \; \text{form} \; \text{method} \; \text{binding} \\
\text{e} &::= \text{event} \; p \; \text{throws} \; \exists \; \text{form} \; \{ \text{contract} \; ] \\
\text{t} &::= \text{e} \; \text{thunk} \; p \\
\text{form} &::= \text{t} \; \text{var.} \; \text{where} \; \text{var} \; \text{this} \\
\text{binding} &::= \text{when} \; p \; \text{do} \; m \\
\text{ep} &::= \text{var} \; \{ \text{new} \; \text{e} \; \text{old} \; \text{e} \} \; \{ \text{e} \; \text{m} \; \{ \text{f} \; \text{e} \} \} \\
\text{es} &::= \{ \text{null} \; \text{var} \; \{ \text{new} \; \text{e} \; \text{old} \; \text{e} \} \; \{ \text{e} \; \text{m} \; \{ \text{f} \; \text{e} \} \} \\
\text{contract} &::= \text{requires} \; p \; \{ \text{assumes} \; \text{se} \} \; \{ \text{ensures} \; \text{ep} \; \text{fep} \} \\
\text{espec} &::= \text{signals} \; \text{ep} \\
\text{spec} &::= \text{requires} \; \text{ep} \; \text{ensures} \; \text{ep} \; \text{throws} \; \exists \\
\text{where} &::= \text{c} \in \text{C}, \text{set of class names} \\
\text{d} &::= \text{C} \cup \{\text{Object}\}, \text{a set of superclass names} \\
\text{p} &::= \mathcal{T}, \text{set of event types names} \\
\text{f} &::= \mathcal{F}, \text{set of field names} \\
\text{var} &::= \{\text{this.next}\} \cup \mathcal{V} \text{is a set of variable names} \\
\text{z} &::= \{\text{X} \cup \{\text{NonNullException, RunTimeException, ClassCastException, NullPointerException, EventHandler}\}, \text{Xis a set of exception names} \\
\end{align*}
\]

**Figure 5: Ptolemy₃’s syntax, based on [1][2]**

### 3.1.1 Declarations

In a class declaration, the class has a name (c), a super class (d), a set of fields (form), methods (meth) and binding declarations (binding). A binding declaration associates an event type (e) with an observer handler method (m). In Ptolemy₃, observers are normal classes with handler methods which take a handler chain as their parameter. Both observer handler methods and non-handler regular methods (meth) have to declare their checked exceptions (ep) in their throws clauses. However, similar to Java, the declaration of unchecked exceptions is not necessary neither for the observer handler methods nor for the regular methods.

In an event type declaration, the event has a name (p), a return type (e), a set of checked boundary exceptions (ep), a set of context variables (form) and an optional translucid contract (contract). The boundary exceptions of the event are checked exceptions, named in its throws clause, that limit the set of checked exceptions that subjects and observers of the event can throw. All the other checked exceptions must be handled locally by the subjects and observers. The boundary exceptions address the problem (l) for checked exceptions. For example, Figure 5 lists a boundary exception RegDExc in the declaration of WithdrawEv, line 29 Declaration of the context variables specify types and names of information shared between the subjects and observers.

### 3.1.2Specifications

The idea is to use the translucid contract of an event to reason about behaviors and flows of the exceptions of its observers independent of the unknown observers or their execution order in their chain. The translucid contract, (contract), is a greybox specification that specifies behaviors of both checked and unchecked exceptions of subject and observers of an event, during its announcement and handling. The contract also specifies the set of unchecked exceptions that subject and observers of the event can throw and the flows of exceptions among the observers of the event in their chain.

**Behaviors of (Un)Checked Exceptions** In a translucid contract, behaviors of checked or unchecked exceptions thrown during announcement and handling of an event is specified using the signals clauses. A signals clause (signals) ep lists checked or unchecked exceptions (ep) of the observers and events of the event and associates them with a side effect free postcondition (ep). The signals clause says if announcement or handling of the event terminates abnormally by throwing any of the exceptions ep then it terminates in a state that satisfies the exceptional postcondition ep. In other words, if an observer or a subject throws an exception ep (ep) then it must satisfy the exceptional postcondition ep. Post-condition of an exception is the state right before the exception is thrown [25]. Exceptions with no exceptional postconditions, have the default postcondition of true. Exceptional postconditions address the problem (2). Figure 6 illustrates the exceptional post-conditions for the checked exception RegDExc, line 9 and the unchecked exceptions RuntimeException, line 10.

The translucid contract also specifies normal behavior of the announcement and handling of an event [1]. The normal behavior requires ep₁ ensures ep₂ says that if the event is announced in a state that satisfies the precondition ep₁ and its announcement and handling terminates normally, i.e. throws no exceptions, then it terminates in a state that satisfies the postcondition ep₂. The specification establishes ep₁ is sugar for requires true ep₁.

An assumes (se) block in a translucid contract limits the implementation structure of the observers of its event. The body (se) of the assumes block is a combination of program expressions and exception-aware specification expressions spec. Program expressions reveal interesting exceptional control flows, e.g. try-catch or invoke expressions in the implementation of the refining observers, whereas exception-aware specification expressions hide the rest of the implementation under refining expressions. Figure 12 illustrates an assumes block in the translucid contract of WithdrawEv, lines 24 with the program expressions on lines 26 and 27 and the exception-aware specification expressions on lines 28 and 29.

### Limiting Unchecked Exceptions

Exception-aware specification expressions in an assumes block limit the set of unchecked exceptions of the subjects and observers of an event, in combination with program expressions. The exception-aware specification expression requires ep₁ ensures ep₂ throws ep₃ says that in a re-finining implementation of an observer, if the execution starts in a state satisfying the precondition ep₁ and terminates normally, it terminates in a state satisfying the postcondition ep₂. However, if the execution terminates abnormally by throwing an unchecked exception, then the thrown exception must an exception in ep₃ otherwise the specification expression is violated. The exception-aware specification expression requires ep₁ ensures ep₂ throws ep₃ RuntimeException. Exception-aware specification expressions address the problem (l) for unchecked exceptions.

For example, Figure 6 illustrates an exception-aware specification expression on lines 48 that allows all unchecked exceptions RTE Exception to be thrown by the observers of WithdrawEv. The program expression next.invoke(), line 6 can throw any unchecked exception since it invokes the next observer of WithdrawEv. Note that the event closure next is
non-null by construction. Figure 11 illustrates another exception-aware specification expression, line \(5\) that allows no unchecked exceptions, i.e. \texttt{NoRuntimeExc}. The program expressions \texttt{next.invoke()} in this figure, line \(3\), cannot throw any unchecked exceptions since it invokes the next observer. Finally, Figure 12 illustrates two exception-aware specification expressions, lines \(4,5\) and \(6\). The first one allows all unchecked exceptions by a refining implementation in an observer of WithdrawEv, however, the program expression try-catch which surrounds it, lines \(3\) and \(7\) catches all such unchecked exceptions and handles them in a way that itself may not throw any unchecked exceptions, line \(8\).

**Flows of (Un)checked Exceptions** An invoke expression causes the invocation of the next observers in a chain of observers. Revealing the invoke expression and its enclosing expression in an assumes block specifies the flows of exceptions of the observers in the chain of the observers. For example, in Figure 4, an observer of WithdrawEv is not allowed to catch any exception thrown by other observers in the chain because there is no try-catch expression around its invoke expression, line \(3\). This means any exception thrown by an observer in the chain propagates back all the way to its subject without being caught by another observer in the chain. In Figure 7, an observer can catch an unchecked exception thrown by another observer in the chain and swallow it because its invoke expression, line \(3\), is enclosed by a try-catch expression, lines \(4\) and \(7\), it cannot catch checked exceptions such as \texttt{RegDExc}. Because invoke expressions play a critical role in understanding the flows of exceptions in the chain of the observers, revealing them in the the assumes block of the translucid contract of the event is mandatory. Revealing invoke expressions in the assumes blocks addresses the problem (3).

In summary, the translucid contract and boundary exceptions of an event limit the set of checked and unchecked exceptions of the subjects and unknown observers of an event. The contract specifies the behaviors of the subjects and observers of the event upon throwing these exceptions and the flows of these exceptions among the observers in their chain. The idea is to use the translucid contract of the event to reason about its announcement and handling independent of its unknown observers or their execution order in the chain. \texttt{Ptolemy}_\texttt{x}'s typing and refinement rules, in Section 4, ensure that the subjects and observers satisfy these specifications.

### 3.1.3 Expressions

\texttt{Ptolemy}_\texttt{x} has expressions for throwing and handling of exceptions and explicit announcement and handling of events. The expressions \texttt{throw} and \texttt{try} \texttt{- catch} are standard and similar to Java [8]. The expressions \texttt{finally} and \texttt{try-catch} expressions with multiple catch clauses are sugars and are omitted from the syntax. \texttt{Ptolemy}_\texttt{x} supports the built-in exceptions \texttt{Throwable}, \texttt{Exception}, \texttt{RuntimeException}, \texttt{NullPointerException} and \texttt{ClassCastException}, similar to exceptions in Java with the same subtyping relations. Unchecked exceptions are subtypes of \texttt{RuntimeException}. The exception \texttt{NoRuntimeExc} stands for no unchecked exception.

In \texttt{Ptolemy}_\texttt{x}, a subject explicitly announces an event \((p)\) using an \texttt{announce} expression with parameters \(\theta\) as its context variables and the event body \((e)\). The announce expression starts with the execution of the event of observers for the event \((p)\). An observer is registered using a \texttt{register} expression that evaluates its parameter \((e)\) to an object and puts it into a chain of observers. Expression \texttt{invoke} evaluates its receiver object \((e)\) into an event closure and runs it, which in turn causes the next observer in the chain of observers to run. An event body is at the end of the chain of the observers. Event closures are also passed to observer handler methods, e.g. line \(10\) of Figure 4. The type \((\texttt{thunk}\ e\ p)\) ensures the value of the corresponding actual parameter is of type event closure with return type \((c)\) for an event \((p)\). The expression \texttt{next} is the placeholder for the event closure.

### 3.2 Modular Reasoning about Behaviors of Exceptions

To enable modular reasoning about behaviors of exceptions in event announcement and handling, e.g. reasoning about \(\Phi_3\) in Section 2.1 \texttt{Ptolemy}_\texttt{x} provides exceptional postconditions. Figure 4 illustrates the exceptional postconditions for the checked boundary exception \texttt{RegDExc}, line \(2\) and unchecked exceptions \texttt{RuntimeException}, line \(10\). The exceptional postcondition for \texttt{RegDExc} says that if during announcement and handling of WithdrawEv an exception \texttt{RegDExc} is thrown by its observers then they must guarantee that the balance of the account \texttt{acc} does not change, i.e. \texttt{acc..bal} == \texttt{old(acc..bal)}.

1. \texttt{event WithdrawEv throws RegDExc{ ...}
2. \texttt{requires acc..null}
3. \texttt{assumes { }
4. \texttt{establishes next.acc().bal==old(next.acc().bal)}
5. \texttt{throws RuntimeException}
6. \texttt{next.invoke();}
7. \texttt{}}
8. \texttt{ensures acc..bal<old(acc..bal)}
9. \texttt{signals (RegDExc) acc..bal==old(acc..bal)}
10. \texttt{signals (RuntimeException) true
11. }}

**Figure 6:** Exception-aware specification expression, lines \(4,5\) and exceptional postconditions, lines \(9,10\).

Using this guarantee one is able to verify the assertion \(\Phi_3\) in a modular fashion using only the implementation of the subject \texttt{Savings}, in Figure 4 and the contract of the event WithdrawEv especially its exceptional postcondition for \texttt{RegDExc}, line \(2\) Recall that the assertion \(\Phi_3\) says if an exception \texttt{RegDExc} is thrown during announcement and handling of WithdrawEv the balance of the account is not changed. Upon announcement of WithdrawEv the control reaches line \(10\) of Figure 4 if an exception \texttt{RegDExc} is thrown by an observer and propagated back to the subject. According to the exceptional postcondition of \texttt{RegDExc} the observers of WithdrawEv ensure that the predicate \texttt{acc..bal == old(acc..bal)} is true upon throwing \texttt{RegDExc}. By replacing the variable \texttt{this} for \texttt{acc} in this exceptional postcondition we get the predicate \texttt{this..bal == old(this..bal)} which is the same as the assertion \(\Phi_3\) that we wanted to verify. Recall that \texttt{this} is passed as the context variable \texttt{acc}, line \(5\) of Figure 4. Such a reasoning is independent of system configuration, i.e. unknown observers of WithdrawEv or their execution order in their chain. This reasoning is valid because \texttt{Ptolemy}_\texttt{x}'s refinement rules require all the observers and subjects of WithdrawEv to satisfy the exceptional postcondition of \texttt{RegDExc} if they throw it. The refinement rules are discussed in Section 4.3.

### 3.3 Modular Reasoning about Flows of Exceptions

To enable modular reasoning about flows of exceptions of event announcement and handling, e.g. reasoning about \(\Phi_{11}\) and \(\Phi_{12}\) in Section 2.2 \texttt{Ptolemy}_\texttt{x} provides exception-aware specification expressions and translucid contracts. The contract in Figure 5 says that a refining observer of WithdrawEv can throw any unchecked exception, lines \(4,5\), however it does not catch any exception thrown by another observers in the chain of observers, line \(6\).
The guarantee that an observer of WithdrawEv does not catch any exception thrown by another observer, means that the property \( \Phi_f \) holds. Recall that \( \Phi_f \) says if an exception RegDEExc is thrown by an observer of WithdrawEv during its announcement and handling, the exception is propagated back to its subject. This guarantee also means that the catch clause \( \text{catch (RuntimeException e) (...) in Savings, line } l \) of Figure 1 is necessary, i.e. \( \Phi_f \), since an unchecked exception thrown by an observer propagates back to the subject. However, if the contract in Figure 1 is used instead of the contract in Figure 2 in reasoning, then such a catch clause will be unused, since the observers refining this contract cannot throw any unchecked exceptions, lines 3–4, and thus no unchecked exceptions reaches the subject.

Again only the implementation of the subject Savings and the contract of event WithdrawEv is enough for reasoning about \( \Phi_f \) and \( \Phi_f \), independent of system configuration, i.e. unknown observers or their execution order in their chain. This is only valid because \( Ptolemy \)’s type system and refinement rules ensure that subjects and observers of an event respect their contract.

### 4. MODULAR REASONING AND REFINEMENT IN Ptolemy

\( Ptolemy \) enables reasoning about behaviors and flows of exceptions of observers using translucid contracts, as illustrated in Section 3.2 and Section 3.3. Such a reasoning is modular and independent of system configuration, i.e. unknown observers or their execution order in their chain, because of the following guarantees:

- each subject and observer of an event only throws the exceptions it is allowed to throw according;
- each subject and observer satisfies the exceptional postconditions of these exceptions upon throwing them; and
- each observer only throws and handles the exceptions at the places specified in its implementation.

These guarantees are provided by \( Ptolemy \)’s type checking rules, refinement rules for static structural refinement and runtime assertion checking of translucid contracts. These refinement rules restrict both subjects and observers of an event and are different from refinement rules in specification languages such as JML [25], although they may look similar syntactically.

#### 4.1 Static Semantics

\( Ptolemy \)’s typing rules ensure that each subject and observer of an event only throws the checked boundary exception they are allowed to. The typing rules also check for structural refinement of the event’s contract that ensures each observer only throws and handles the exceptions at the specified places in its implementation. Previous work on join point interfaces [2] informally discusses the typing rules for boundary-like exceptions, however, it does not provide any formalization or soundness proof.

##### 4.1.1 Type Attributes

\( Ptolemy \)’s typing rules use the type attributes of Figure 7, in which regular types are augmented with a set of checked exceptions \( \tau \).

The type attribute \( \text{exp} \ t, \tau \) denotes expressions of type \( t \) that may throw the checked exceptions \( \tau \). Variables do not throw exceptions and are denoted by \( \text{var} \ t \). The type attribute OK is used to type check top level declarations whereas OK in \( e \) is used for type checking of lower level declarations in the context of an upper level declaration \( e \). In \( Ptolemy \), similar to Java, a checked exception is a subtype of \( \text{Exception} \) that is not a subtype of \( \text{RuntimeException} \). The auxiliary function \( \text{isChecked} \) checks if an exception is a checked exception. The typing rules and auxiliary functions use a fixed class table \( CT \) which is a list of event type and class declarations. We require distinct top level names and acyclic inheritance relations for classes in \( CT \). The typing judgement \( \Gamma \vdash e : \theta \) says that in the typing environment \( \Gamma \) and store typing \( \Pi \) the expression \( e \) has the type \( \theta \). Figure 8 shows select typing rules for \( Ptolemy \). The full set of typing rule could be found in Section 4 in the appendix.

#### 4.1.2 Declaration Typing Rules

One may assume that subjects and observers of an event can throw any subtypes of its checked boundary exceptions. However as observed in previous work on JPIs [2], the observers can throw any subtype of the boundary exceptions and the subjects can only throw them invariably. This is to avoid situations in which an observer may throw an exception that its subjects cannot handle, e.g. for a boundary exception \( E_e \), if a subject throws \( E_s \) and an observer throws \( E_r \), such that \( E_s <: E_r <: E_e \), then the subject cannot handle \( E_r \). The observers can throw any subtypes of the boundary exceptions. Such a relation is denoted using \( \subseteq \) which combines subtype and subset relations.

The rule (T-Binding) checks an observer of the event \( p \) and its binding declaration. The rule checks for the relation \( \subseteq \): between the checked exceptions \( \{ \tau \} \) of the observer handler method \( m \) of the event \( p \) and its boundary exceptions \( \{ \tau \} \), i.e. \( \{ \tau \} \subseteq \{ \tau \} \). Similar to non handler methods, the observer handler method \( m \) must declare its checked exceptions \( \{ \tau \} \). The rule (T-Binding) also ensures that the body \( e \) of the observer handler method \( m \) structurally refines the body \( se \) of the assumes block of its translucid contract of \( \{\text{contract}\} \), i.e. \( se \subseteq e \). The structural refinement \( \subseteq \) is discussed in Section 4.2.

The rule (T-EventTv) can type checks declaration of the event \( p \). The body \( se \) of the assumes block in the translucid contract of \( p \) specifies the implementation structure of its observers. This means, the same \( 
\subseteq \) relation between checked exceptions of an observer and the boundary exceptions of its event must exist between the checked exceptions \( \{ \tau \} \) of the assumes block \( se \) and the boundary exceptions \( \{ \tau \} \) of its event \( p \), i.e. \( \{ \tau \} \subseteq \{ \tau \} \). The type \( t' \) of \( se \) should also be a subtype \( \subset \) of the return type \( e \) of the event.

#### 4.1.3 Expression Typing Rules

Similar to the rule (T-Binding) that type checks observers of an event \( p \), the rule (T-ANNOUNCE) type checks its subjects and especially their announce expressions. The subjects can only throw exact boundary exceptions invariably. This means that the set of the checked exceptions \( \{ \tau \} \) of the subject must be in a subset relation \( \subseteq \) with the boundary exceptions \( \{ \tau \} \) of the event \( p \), i.e. \( \{ \tau \} \subseteq \{ \tau \} \). The relation \( \subseteq \) is different from \( \subseteq \) since it does not allow the sub-
project to throw subtypes of the boundary exceptions. The set of the checked exceptions \( (\mathcal{E}) \) is the union of the set of checked exceptions \( (\mathcal{E}) \) for the body (e) of its announce expression and the set of checked exceptions \( (\mathcal{E}) \) for its parameters \( (e_i) \), i.e., \( \mathcal{E} = \mathcal{E}_e \cup \mathcal{E}_i \). The rule also checks that the types \( (t_i) \) of the parameters \( (e_i) \) of the announce expression are subtypes of the types \( (t'_i) \) of the context variables of the event. The same should hold for the type \( (t'_i) \) of the body (e) of the announce expression and the return type \( (c) \) of the event (p).

The rule (T-INVOKED) checks an invoke expression. The invoke expression invokes the next observer of an event in the chain of its observers. The chain of the observers is included in its event closure receiver object (e). The event closure for an event (p) with return type \( (e) \) and the boundary exceptions \( (\mathcal{E}) \) is of type \( \text{thunk} \in p.\mathcal{E} \). The rule checks for the equality of the set of checked exceptions \( (\mathcal{E}) \) of the invoke expression and the set of the boundary exceptions \( (\mathcal{E}) \) of its event (p), i.e., \( \mathcal{E} = \mathcal{E}_e \). The rule (T-REGISTER) says that the set of checked exceptions \( (\mathcal{E}) \) of a register expression is set of checked exceptions of its parameter \( (e) \).

The rule (T-SPEC) checks an exception-aware specification expression. It checks that the side effect free pre- and postconditions \( (ep_1) \) and \( (ep_2) \) throw no checked exceptions and the exception set \( (\mathcal{E}) \) listed in its throws clause are unchecked exceptions, i.e. subtype of RuntimeException. The exception-aware specification expression throws no checked exceptions because it is a specification expression and has the bottom type \( \perp \) that is a subtype of any other type. The rule (T-REFINING) checks a refining expression that refines an exception-aware specification expression (spec). The rule simply says that the set of checked exceptions \( (\mathcal{E}) \) of the refining expression is the same as the set of checked exceptions of its body \( (e) \).

The rule (T-THROW) checks a throw expression. It adds the throw exception \( (t) \) to the exception set \( (\mathcal{E}) \) of the expression, if the exception is a checked exception. The conditional isChecked \( (t) ? \mathcal{E} = \{ t \} : \mathcal{E} = \emptyset \) checks if \( (t) \) is a checked exception. Recall that the typing rules only keep track of the checked exceptions and not unchecked exceptions. The rule (T-TRYCATCH) checks a try-catch expression. It computes the set of checked exceptions \( (\mathcal{E}) \) of the body \( (e_1) \) of the try part and then factors out checked exceptions that are subtypes of the exception \( (x) \) handled by the catch part. The result is the set of the checked exceptions \( (\mathcal{E}) \) that are thrown by the try part and not handled in the catch part. The body of the catch part itself can throw the checked exceptions \( (\mathcal{E}) \). The set of the checked exceptions of the try-catch expression is the union of \( (\mathcal{E}) \) and \( (\mathcal{E}) \). The rest of the expression typing rules in Section 3.8 compose the set of checked exceptions thrown by an expression.

4.1.4 Soundness of Type System

Ptolemy’s type system is proven sound following standard progress and preservation arguments. Treatment of features such as exception-aware specification expressions, translucid contracts and refining expressions are new in the dynamic semantics and the proof. The proof of soundness and the full set of dynamic and static semantics rules could be found in Section 3.8.

4.2 Structural Refinement

Structural refinement rules ensure that each observer of an event only throws and handles the exceptions at the places in its implementation as specified by the assumes block of its contract. Figure 8 shows select rules for Ptolemy’s structural refinement. For an observer of the event (p), the implementation \( (e) \) of its observer handler method \( (m) \) structurally refines the assumes block \( (se) \) of its translucid contract, i.e. \( se \subseteq e \), if the following holds: for each program expression in \( (se) \) there is a textually match program expression in \( (e) \) at the same place, e.g. the rule var for the variable expressions. And for each exception-aware specification expression \( spec \in (se) \) there is a refining expression \( refining spec(e) \) at the same place in the implementation which claims to refine \( spec \). The rule for either \( (se_1) \) or \( (se_2) \) allows the observer to either refine \( (se_1) \) or \( (se_2) \) and thus allow variability in the observers. Structural refinement for other expressions is based on the refinement of their subexpressions. For example, the try-catch expression \( \text{try} \{ se_1 \} \text{catch} \{ x \text{var} \} \{ se_2 \} \) is structurally refined by...
try\{e_1\}catch(x\;var)\{e_2\}

4.3 Runtime Assertion Checking

Structural refinement rules ensure that for each exception-aware specification expression \(\text{spec}\) of the form \text{requires} \(e_1\) \text{ensures} \(e_2\) \text{throws} \(T\) in a transduced contract of an event, there is a refining expression \text{refining} \(\text{spec}\) which claims to refine \(\text{spec}\), in the implementation of the observers of the event. However, they do not statically check this claim that the body \(e\) of the refining expression only throws the specified unchecked exceptions \(T\). In Ptolemy\(_\chi\), runtime assertion checking (RAC) does this check. RAC also ensures that the observers and subjects of an event satisfy the exceptional postconditions of their exceptions when they throw them. Figure 9 illustrates, in grey, the runtime assertion checking for the observer handler method \text{check} of Figure 8 with its transduced contract in Figure 6. The code for RAC is generated automatically by the compiler.

In Figure 10 a try-catch expression surrounds the original body of the check to catch the checked and unchecked exceptions that the observer is allowed to throw, that is \text{RegDExc}, lines 22–25 and \text{RuntimeException}, line 26–29 and to check for their exceptional postconditions, lines 24 and 28. After checking for the exceptional postconditions, the exceptions are rethrown to avoid changing of the exceptional control flow of the program, lines 25 and 29. The method \text{Con.signal} checks for the exceptional postcondition of the exception \(e\) and aborts the execution or raises an error of type \text{Error} in case of their violation.

There is another try-catch that surrounds the refining expression, lines 7–15. This try-catch expression catches all unchecked exceptions of the refining expression and checks if they are allowed to be thrown according to the exception-aware specification expression that the refining expression claims to refine, lines 8–12. The method \text{Con.allowedExc} checks if the thrown exception \(e\) is among the set of allowed exceptions and aborts the program if it is not, otherwise it rethrows the exception. In this example the exception-aware specification expression allows all unchecked exceptions to be thrown by its refining expression, i.e. \text{throws} \text{RuntimeException} on line 9. RAC also checks for the pre- and normal postcondition of the observer, lines 3 and 20 and the pre- and normal postcondition of the refining expression, lines 6 and 17 using the methods \text{Con.require} and \text{Con.ensure}.

5 EVALUATION

In this section, usability of Ptolemy\(_\chi\)'s exception flow reasoning is evaluated by using it to understand (non-) occurrence of a set of bug patterns [6] for aspect-oriented (AO) programs. The overhead of the application of Ptolemy\(_\chi\) to a simple Ptolemy program is also discussed. Understanding the bug patterns requires knowledge about the set of checked and unchecked exceptions of unknown observers and flows of these exceptions in the chain of the observers.

5.1 AO Bug Patterns in Ptolemy\(_\chi\)

Despite their differences [12], Ptolemy\(_\chi\) and AO languages such as AspectJ share some similarities. In these languages a subject (base code) announces an event implicitly or explicitly, observers (aspects) register for the event and are invoked implicitly and run in a chain upon announcement of the event. This in turn suggests that there may be similarities between bug patterns of these languages. Coelho et al. [6] introduces a set of AO bug patterns especially with regard to exception handling. In this section we adapt these bug patterns to Ptolemy\(_\chi\) for our running bank account example and show how their occurrence or non occurrence could be understood using Ptolemy\(_\chi\)'s exception flow reasoning.

Coelho et al. [6] recognize 5 category of bug patterns in AO: (i) throw without catch (ii) residual catch (iii) exception stealer (iv) path dependent throw and (v) fragile catch. The last two bug patterns are not applicable to Ptolemy\(_\chi\). A path dependent throw bug occurs when exceptions that a method throws vary based on different call chains leading to its invocation. This could happen because of AO scope designator constructs, such as \text{with} and \text{eflow} in AspectJ, that are not supported in Ptolemy\(_\chi\). A fragile catch bug occurs when an aspect is supposed to catch an exception in a specific program point but misses it due to a problematic pointcut, i.e. pointcut fragility. Pointcut fragility does not happen in Ptolemy\(_\chi\), because of its explicit event announcement [32].
5.2 Throw Without Catch

In Ptolemy, a throw without catch bug happens when an observer throws an exception during announcement and handling of an event and there is no handler to catch it, neither on the observer nor the subject side. This is especially true for unchecked exceptions since checked exceptions cannot go uncaught.

**Occurrence**

The subject Savings in Figure 1 and the contract in Figure 2 together illustrate a throw without catch bug if the catch (RuntimeException e) {...} line of Figure 3 is inadvertently forgotten by the developer. Similar to Java, Ptolemy does not complain about the missing catch clause for unchecked exceptions. The contract for WithdrawEv in Figure 4 allows the observers of WithdrawEv to throw any unchecked exception, lines 3-5. It also does not allow an observer in the chain to catch any exceptions thrown by another observer of WithdrawEv since there is no try-catch surrounding the invoke expression, line 6. Thus if an observer of WithdrawEv throws an unchecked exception the exception propagates back to the subject Savings which also does not catch the exceptions, i.e. the exception goes uncaught during announcement and handling of WithdrawEv. One may argue that leaving the catch clause catch (RuntimeException e) {...} in its place on line 3 of Figure 4 avoids the throw without catch bug, however, this catch clause itself could be unnecessary and unused and a residual bug if the observers of WithdrawEv do not throw any unchecked exceptions.

**Non Occurrence**

To ensure non occurrence of a throw without catch bug during announcement and handling of an event, one may want to force the observers of the event to not throw any unchecked exceptions. Figure 11 illustrates a variation of the contract of Figure 2 in which the observers of WithdrawEv cannot throw any unchecked exceptions. The exception-aware specification expression on lines 3-6 limits the set of unchecked exceptions of the observers of Withdraw to nothing, i.e. throws NoRuntimeException.

```java
1 void event WithdrawEv throws RegDExc{ ..
2 assumes{
3 establishes next.acc().bal==old(next.acc()).bal
4 throws NoRuntimeException;
5 next.invoke();
6 } .. }
```

Figure 11: No unchecked exceptions for observers, lines 3-6.

Figure 12 illustrates another variation of the contract for WithdrawEv in which each observer has a try-catch expression in its body, lines 6 and 7. The try-catch expressions catch any unchecked exception that might be thrown by the observers and handles them in a way that does not throw any unchecked exception itself, line 8.

```java
1 void event WithdrawEv throws RegDExc{ ..
2 assumes{
3 try{
4 establishes next.acc().bal==old(next.acc()).bal
5 throws RuntimeException;
6 next.invoke();
7 } catch (RuntimeException e){
8 establishes true throws NoRuntimeException; }
9 } .. }
```

Figure 12: Observers handle their unchecked exceptions locally, lines 6 and 7 and 8.

Unlike the contract in Figure 1, the contract in Figure 2 requires the implementation of the observers of WithdrawEv to be surrounded by the specified try-catch expression. It also allows an observer to catch and swallow the unchecked exceptions thrown by other observers in the chain which is not allowed in Figure 1.

5.3 Residual Catch

In Ptolemy, a residual catch bug happens when a subject tries to catch an exception that is not thrown or is already caught by its observers before reaching it.

**Occurrence**

The subject Savings in Figure 1 and the contract in Figure 2 together illustrate the occurrence of a residual catch bug. The subject Savings catches the unchecked exceptions thrown by its observers during announcement and handling of WithdrawEv, line 3. However, the contract for WithdrawEv in Figure 11 does not allow its observers to throw any unchecked exceptions, lines 5-6. This in turn means the catch clause on line 7 is not necessary since the subject is trying to catch exceptions that are not thrown by its observers. Recall that in Ptolemy, the contract limits the set of exceptions of both observers and the subject, Savings do not throw any unchecked exceptions, however the subject is still trying to catch the throw without catch bug by a subject is caught by another observer action where an exception is caught wrongly by an unintended handler. In Ptolemy, a variation of this bug happens when an exception thrown by an observer that was supposed to be caught by a subject is caught by another observer in the chain.

**Non Occurrence**

A residual bug does not occur during announcement and handling of WithdrawEv in the subject Savings, Figure 1 if its observers do not throw any unchecked exceptions and its catch clause, lines 3-6 is omitted. The subject Savings without the catch (RuntimeException e) {...} and the contract in Figure 11 illustrate non occurrence of a residual catch bug. The contract for WithdrawEv in Figure 11 does not allow its observer to throw any unchecked exception and the subject does not catch any unchecked exceptions and there is no unnecessary catch clause. The same is true for the contract in Figure 2, since it does not allow its observers to throw unchecked exceptions too. The subject Savings with the catch clause catch (RuntimeException e) {...} on line 3 and the contract in Figure 11 illustrate another example of the non occurrence of a residual bug. Here the contract allows the observers of WithdrawEv to throw any unchecked exceptions that could be propagated to the subject. This in turn means that the catch clause in the subject is actually necessary and used.

5.4 Exception Stealer

An exception stealer bug happens when an exception that was supposed to be caught by an exception handling observer is caught by a subject. This bug pattern is a special case of an unintended handler action where an exception is caught wrongly by an unintended handler. In Ptolemy, a variation of this bug happens when an exception thrown by an observer that was supposed to be caught by a subject is caught by another observer in the chain.

**Occurrence**

The subject Savings in Figure 1 and the contract for its WithdrawEv in Figure 3 illustrate the occurrence of an exception stealer bug. According to the property \( \Phi_1 \) of the bank account example, Section 4.2, if an exception RegDExc is thrown during announcement and handling of WithdrawEv it is propagated back to the subject to be handled. However, the try-catch expression of the contract, lines 4-5 that surrounds the invoke expression allows an observers of WithdrawEv to catch any exception, including RegDExc, if thrown by another observer in the chain and swallow it, line 6, before it reaches the subject Savings.

**Non Occurrence**

The subject Savings in Figure 1 and the contract in Figure 3 illustrate non occurrence of the exception stealer bug for the exception RegDExc. The contract for WithdrawEv in Figure 5 does not allow any try-catch expression to surround the
invoke exceptions in the implementation of its observers. This in turn means if an exception RegDExc is thrown by an observer, it is not caught by other observers in the chain and is propagated back to the subject. Similarly the contract in Figure 7 does not allow the checked exception RegDExc thrown by one observer to be stolen by another observer and propagates it back to Savings. However, unchecked exceptions thrown by one observer could be caught by another observer in the chain, line 7.

5.5 Summary of Bug Patterns

Out of the three Coelho et al. ’s bug patterns that are applicable to Ptolemy, (non) occurrence of all of them could be understood using Ptolemy’s exception reasoning technique. The key in reasoning about these patterns is to know about the set of checked and unchecked exceptions of subjects and observers of an event and especially flow of these exceptions in the chain of the observers. These patterns in our running bank account example are understood using only the transfluc contract of WithdrawEv.

<table>
<thead>
<tr>
<th>Bug Pattern</th>
<th>Occurrence</th>
<th>Non Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throw without catch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Residual catch</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Exception stealer</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Path dependent throw</td>
<td>Not applicable to Ptolemy</td>
<td></td>
</tr>
<tr>
<td>Fragile catch</td>
<td>Not applicable to Ptolemy</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: Stealing and swallowing unchecked exceptions through subsumption, line 6.

5.6 Application to Ptolemy

In this section we discuss the overhead of the application of Ptolemy to a simple figure editor Ptolemy program, that is shipped with its compiler distribution. The figure editor allows creation of simple figures such as points and lines. An event is announced when a figure moves and an observer updates the screen by invoking a mock method. The example has 1 event, 1 subject and 1 observer, 7 classes in total with 127 lines of code. One requirement for the figure editor could be that the observers of the event do not throw any checked exception and their unchecked exceptions must be propagated back to the subject for handling.

To specify this requirement a simple transfluc contract requires true assumes { next.invoke(); establishes true throws RuntimeException; } ensures true should be added to the event declaration. Ptolemy’s event declarations do not have any throws clause which is interpreted as no checked boundary exceptions in Ptolemy. The observer of the event should have a refining expression refining establishes true throws RuntimeException{...} in its implementation to structurally refine the contract. The contract and the refining expression adds 8 lines of the code to the program which increases its size by 6.2%. However, the compiler could be easily modified to not require the default pre- and postconditions of requires true and ensures true and insert the refining expression automatically based on the structural refinement rules. Thus the real overhead in terms of lines of code written by the programmer is only 4 lines of code, i.e. about 3.1%. The programmer only needs to write the contract in the event type declaration. This overhead varies for different programs depending on number of events, observers, etc.

6. RELATED WORK

Reasoning in Implicit Invocation (II) Garlan et al. [14] propose a compositional II reasoning technique to reason about normal behavior by breaking down a system into independent subsystems and relying on system configuration such as number of events, their consumption policy, etc. Dingel et al. [7] propose a rely-guarantee-like reasoning technique by relying on sound and complete announcement of semantic-carrying events, and invariants weakened by location predicates. Krishnaswami et al. [24] verify the structure and normal behavior of the Observer design pattern using separation logic by relying on system configuration such as number of observers and their individual invariants. Our work is focused on modular reasoning about behaviors and flows of exceptions for abnormal termination of event announcement with chained observers, independent of system configuration.

Join point interfaces (JPIs) [2] enable modular type checking for an aspect-oriented (AO) language in the presence of exceptions. Khatchadourian et al. [20] propose a rely-guarantee-like technique to reason about normal behaviors and traces of AO programs. Crosscutting programming interfaces (XPIs) [38] and Join Point Types (JPTs) [27] allow informal specification and reasoning about normal behaviors of aspects in AO. Pipes [42] enables global reasoning about normal and exceptional behaviors in AspectJ. Execution levels [10] use level shifting operations to specify the flow and interaction of exceptions of aspects and base code in AspectJ to avoid exception conflation. Our proposal is mostly focused on reasoning about both exceptional behaviors and flows of exceptions in a non-global and modular fashion.

Reasoning in Explicit Invocation (EI) Supertype abstraction [26] and ESC/Java [12] allow modular reasoning about normal and exceptional behaviors using specifications in JML. Jacobs et al. [16] propose a modular verification technique for invariants and locking patterns in the presence of unchecked exceptions in multithreaded programs. Failboxes [17] provide exceptional behavior guarantees in terms of dependency safety. Anchored exceptions [10] and the work of Malayeri and Aldrich [29] enable modular reasoning about flow of exceptions using more expressive exception interfaces. EJFlow [11] enables reasoning about exception flow using architectural level exception ducts. Jex [34], and the works of Jo et al. [18], Sinha et al. [56] and Fu et al. [13] propose global flow analyses for exceptions in Java. Leroy and Pessaux [27] propose a type based program analysis to estimate the set of uncaught exceptions in ML. Abnormal types [9] guarantee types of exceptions upon abnormal termination. As discussed in Section 1, these techniques use known invocation relations among modules of a system and are not concerned about II languages in which a subject can invoke unknown observers and run them in a chain.

Reasoning Using Greybox Specifications Tyler and Soundarajan [59] and Shaner et al. [55] use greybox specifications for verification of mandatory calls. Rajan et al. [33] use greybox specification to reason about web service policies.

7. CONCLUSIONS AND FUTURE WORK

Modular reasoning about behaviors and flows of exceptions faces unique challenges in event-based implicit invocation (II) languages such as Ptolemy that allow subjects to invoke unknown
observers and run them in a chain. In this work we have illustrated these challenges in Ptolemy and proposed Ptolemy's exception-aware specification expressions, boundary exceptions, exceptional postconditions along with greybox contracts allow limiting the set of exceptions that subjects and observers of an event may throw and specifying behaviors and flows of these exceptions. Ptolemy's sound type system, static structural refinement and runtime assertion checks enable its modular reasoning independent of unknown observers of an event or their execution order in the chain of observers.

For future work, the first task would be a precise formalization of the relation between refinement [31] and structural refinement for abnormal termination. For normal termination structural refinement implies refinement [135]. The second task would be application of Ptolemy to similar languages and evaluate its robustness.

Acknowledgements This work was supported in part by NSF grant CCF-10-17334. We thank Rex D. Fernando, Robert Dyer and the anonymous reviewers of AOSD’13.

APPENDIX

A. DYNAMIC SEMANTICS

In this section we discuss the substitution-based small step operational semantics of Ptolemy with the focus on exception-aware specification expressions and throwing and handling of exceptions.

A.1 Dynamic Semantics Objects

Ptolemy's operational semantics relies on three additional expressions loc, raise loc and evalpostexc e ep τ that are not part of its surface syntax. The expression loc represents a location in the store. The expression raise loc denotes throwing of an exception loc and is used in exception propagation. The expression evalpostexc e ep τ is used in enforcing an exception-aware specification expression at runtime, especially for the refining expression which claims to refine it. Figure 15 shows the added syntax for these expressions. It also shows the evaluation contexts and configuration used in Ptolemy's operational semantics.

In Figure 15 an evaluation context E specifies the evaluation order of an expression and the position in the expression where the evaluation is happening. Ptolemy uses a left-most innermost call-by-value evaluation policy. Evaluation contexts, in combination with raise expressions, are also used for exception propagation. Basically if an exception is thrown in an evaluation context of an expression, except the try-catch expression, then the whole expression evaluates to a raise expression immediately, e.g. [E[| τ var = raise loc: e] , S, Π, A] ⇔ [E[| raise loc], S, Π, A]. This in turn enables propagation of the exception by unwinding call stack until the exception reaches the nearest enclosing try-catch expression or the program entry [8]. Figure 18 shows Ptolemy's operational semantics rules for exception propagation. Exception propagation in Ptolemy is similar to Java.

Ptolemy's operational semantics transitions from one configuration to another. Figure 15 shows a configuration (E) which contains an expression (e), store (S), store typing (Π) and an ordered list of active observers (A). The store typing is only used for soundness proof, although it is maintained and updated by the dynamic semantic rules. A value (v) is either a location (loc) or (null). The store maps locations to storables (sv) which are either objects records (o) or event closures (ec). An object record has a class name (c) and a map (F) from fields to values. An event closure (eclosure(H,e,p)) contains an ordered list of observer handlers (H), an expression (e) and an environment (p). An observer handler method (h) contains a location (loc) which points to an observer object and a method name (m).

A.2 Throwing and Handling of Exceptions

Ptolemy's mechanism for throwing and handling of exceptions is similar to Java [8], i.e. throw expressions throw exceptions and try – catch expressions handle them if their catch clauses are applicable. Unlike Java, Ptolemy also supports exception-aware specification expressions and refining expressions which claim to refine the exception-aware specification expressions. Figure 16 shows normal operational semantics of these expressions and Figure 17 shows their exceptional semantics. An execution is normal if every exception throw is handled.

---

In Figure 15 an evaluation context E specifies the evaluation order of an expression and the position in the expression where the evaluation is happening. Ptolemy uses a left-most innermost call-by-value evaluation policy. Evaluation contexts, in combination with raise expressions, are also used for exception propagation. Basically if an exception is thrown in an evaluation context of an expression, except the try-catch expression, then the whole expression evaluates to a raise expression immediately, e.g. [E[| τ var = raise loc: e] , S, Π, A] ⇔ [E[| raise loc], S, Π, A]. This in turn enables propagation of the exception by unwinding call stack until the exception reaches the nearest enclosing try-catch expression or the program entry [8]. Figure 18 shows Ptolemy's operational semantics rules for exception propagation. Exception propagation in Ptolemy is similar to Java.

Ptolemy's operational semantics transitions from one configuration to another. Figure 15 shows a configuration (E) which contains an expression (e), store (S), store typing (Π) and an ordered list of active observers (A). The store typing is only used for soundness proof, although it is maintained and updated by the dynamic semantic rules. A value (v) is either a location (loc) or (null). The store maps locations to storables (sv) which are either objects records (o) or event closures (ec). An object record has a class name (c) and a map (F) from fields to values. An event closure (eclosure(H,e,p)) contains an ordered list of observer handlers (H), an expression (e) and an environment (p). An observer handler method (h) contains a location (loc) which points to an observer object and a method name (m).

A.2 Throwing and Handling of Exceptions

Ptolemy’s mechanism for throwing and handling of exceptions is similar to Java [8], i.e. throw expressions throw exceptions and try – catch expressions handle them if their catch clauses are applicable. Unlike Java, Ptolemy also supports exception-aware specification expressions and refining expressions which claim to refine the exception-aware specification expressions. Figure 16 shows normal operational semantics of these expressions and Figure 17 shows their exceptional semantics. An execution is normal if every exception throw is handled.

---

In Figure 15 an evaluation context E specifies the evaluation order of an expression and the position in the expression where the evaluation is happening. Ptolemy uses a left-most innermost call-by-value evaluation policy. Evaluation contexts, in combination with raise expressions, are also used for exception propagation. Basically if an exception is thrown in an evaluation context of an expression, except the try-catch expression, then the whole expression evaluates to a raise expression immediately, e.g. [E[| τ var = raise loc: e] , S, Π, A] ⇔ [E[| raise loc], S, Π, A]. This in turn enables propagation of the exception by unwinding call stack until the exception reaches the nearest enclosing try-catch expression or the program entry [8]. Figure 18 shows Ptolemy’s operational semantics rules for exception propagation. Exception propagation in Ptolemy is similar to Java.

Ptolemy’s operational semantics transitions from one configuration to another. Figure 15 shows a configuration (E) which contains an expression (e), store (S), store typing (Π) and an ordered list of active observers (A). The store typing is only used for soundness proof, although it is maintained and updated by the dynamic semantic rules. A value (v) is either a location (loc) or (null). The store maps locations to storables (sv) which are either objects records (o) or event closures (ec). An object record has a class name (c) and a map (F) from fields to values. An event closure (eclosure(H,e,p)) contains an ordered list of observer handlers (H), an expression (e) and an environment (p). An observer handler method (h) contains a location (loc) which points to an observer object and a method name (m).

A.2 Throwing and Handling of Exceptions

Ptolemy’s mechanism for throwing and handling of exceptions is similar to Java [8], i.e. throw expressions throw exceptions and try – catch expressions handle them if their catch clauses are applicable. Unlike Java, Ptolemy also supports exception-aware specification expressions and refining expressions which claim to refine the exception-aware specification expressions. Figure 16 shows normal operational semantics of these expressions and Figure 17 shows their exceptional semantics. An execution is normal if every exception throw is handled.

---

In Figure 15 an evaluation context E specifies the evaluation order of an expression and the position in the expression where the evaluation is happening. Ptolemy uses a left-most innermost call-by-value evaluation policy. Evaluation contexts, in combination with raise expressions, are also used for exception propagation. Basically if an exception is thrown in an evaluation context of an expression, except the try-catch expression, then the whole expression evaluates to a raise expression immediately, e.g. [E[| τ var = raise loc: e] , S, Π, A] ⇔ [E[| raise loc], S, Π, A]. This in turn enables propagation of the exception by unwinding call stack until the exception reaches the nearest enclosing try-catch expression or the program entry [8]. Figure 18 shows Ptolemy’s operational semantics rules for exception propagation. Exception propagation in Ptolemy is similar to Java.

Ptolemy’s operational semantics transitions from one configuration to another. Figure 15 shows a configuration (E) which contains an expression (e), store (S), store typing (Π) and an ordered list of active observers (A). The store typing is only used for soundness proof, although it is maintained and updated by the dynamic semantic rules. A value (v) is either a location (loc) or (null). The store maps locations to storables (sv) which are either objects records (o) or event closures (ec). An object record has a class name (c) and a map (F) from fields to values. An event closure (eclosure(H,e,p)) contains an ordered list of observer handlers (H), an expression (e) and an environment (p). An observer handler method (h) contains a location (loc) which points to an observer object and a method name (m).

---

In Figure 15 an evaluation context E specifies the evaluation order of an expression and the position in the expression where the evaluation is happening. Ptolemy uses a left-most innermost call-by-value evaluation policy. Evaluation contexts, in combination with raise expressions, are also used for exception propagation. Basically if an exception is thrown in an evaluation context of an expression, except the try-catch expression, then the whole expression evaluates to a raise expression immediately, e.g. [E[| τ var = raise loc: e] , S, Π, A] ⇔ [E[| raise loc], S, Π, A]. This in turn enables propagation of the exception by unwinding call stack until the exception reaches the nearest enclosing try-catch expression or the program entry [8]. Figure 18 shows Ptolemy’s operational semantics rules for exception propagation. Exception propagation in Ptolemy is similar to Java.

Ptolemy’s operational semantics transitions from one configuration to another. Figure 15 shows a configuration (E) which contains an expression (e), store (S), store typing (Π) and an ordered list of active observers (A). The store typing is only used for soundness proof, although it is maintained and updated by the dynamic semantic rules. A value (v) is either a location (loc) or (null). The store maps locations to storables (sv) which are either objects records (o) or event closures (ec). An object record has a class name (c) and a map (F) from fields to values. An event closure (eclosure(H,e,p)) contains an ordered list of observer handlers (H), an expression (e) and an environment (p). An observer handler method (h) contains a location (loc) which points to an observer object and a method name (m).
The rule `(throw)` evaluates a throw expression to a raise expression which is used for exception propagation and handling. However, if the parameter of the throw expression evaluates to `null` then a `NullPointerException` is thrown in the rule `(X-Throw)`. An exception could be caught and handled by a try-catch expression. In the rule `(R-TryCatch)`, evaluation of the try part throws an exception, i.e., it evaluates to a raise loc. Because the type `(c)` of the thrown exception is a subtype of the exception `(x)` that the catch part handles, i.e., `c < x`, the whole try-catch expression evaluates to `(e')` which is the catch part `(e)` with the exception variable `(var)` substituted with the thrown exception `(loc)`. If the catch part can not handle the exception, i.e., `!(c <: x)`, then the exception is propagated by reducing the whole try-catch expression to a raise expression in the rule `(X-TryCatch)`. `S(loc)` returns the object record `[c.F]` for the location `loc` in the store in which `c` is the type of the object. If the try part of the try-catch expression does not throw any evaluation, i.e., it evaluates to a value `(v)`, then the whole try-catch expression is evaluated to `(v)` in the rule `(Try-Catch)`. The rule `Refining` ensures that a refinement expression which claims to refine an exception-aware specification expression actually refines the exception-aware specification expression, i.e., its body `(e)` only throws the unchecked exceptions `(∀)` that are allowed by the refining expression. It also checks if the pre- and postconditions of the refining expression are satisfied by `(e)`. These checks are illustrated in Section 4.2. The rule `[Refining]` first checks if its precondition `(n)` holds, i.e., `n ≠ 0`. If the precondition holds then the refining expression evaluates to an `evalpostexc` expression, otherwise it evaluates to `↓` in the rule `[X-Refining]` in Ptolemy. `↓` marks violation of a contract which results in halting of the evaluation. The rule `[EvalPostExc-1]` checks for the postcondition `(n)` of the refining expression to hold otherwise it evaluates to `↓` in the rule `[X-EvalPostExc-1]`. The refining expression can only throw unchecked exceptions `(∀)` if its evaluation throws an exception. The rule `[EvalPostExc-2]` checks to ensure any unchecked `(loc)` of type `(c)` that is thrown during the evaluation of the refining expression is actually an allowed exception by the exception-aware specification expression which the refining expression claims to refine i.e., `{c} ⊆ ℙ`. If `(c)` is not an allowed exception, i.e., `!(c) ⊆ ℙ`, then the evaluation halted in the rule `[X-EvalPostExc-2]`.

(X-Throw)

θ(throw null, S, Π, A) → θ(throw new NullPointerException[], S, Π, A)

(X-TryCatch)

θ(try [ raise loc ] catch [ x var ] [ {e} ], S, Π, A) → θ(raise loc, S, Π, A)

(X-Refining)

n == 0

θ(refining requires n ensures cp throws {e}, S, Π, A) → ↓, S, Π, A

(X-EvalPostExc-1)

n == 0

θ(evalpostexc v n [F], S, Π, A) → ↓, S, Π, A

(X-EvalPostExc-2)

θ(evalpostexc [raise loc] n [F], S, Π, A) → ↓, S, Π, A

Figure 17: Exceptional operational semantics of Ptolemy's select expressions, based on [8, 32].

Similar to Ptolemy [32], Ptolemy supports explicit announcement and handling of typed events in addition to standard object-oriented (OO) expressions. Figure 18 shows the operational semantics rules of these expressions. Figure 19 shows the operational semantics of Ptolemy's OO expressions. And Figure 20 shows the auxiliary functions used in these rules. Descriptions of these rules could be found in our previous work on Ptolemy [32].

P-Invoke

θ(throw loc. invoke [], S, Π, A) → θ(throw loc, S, Π, A)

P-Register

θ(register(throw loc)], S, Π, A) → θ(throw loc, S, Π, A)

P-Annotate

θ(announce p(throw loc e..)], S, Π, A) → θ(throw loc, S, Π, A)

P- milf

θ(throw loc.m(r)], S, Π, A) → θ(throw loc, S, Π, A)

P-Call-1

θ(throw loc.s(m)], S, Π, A) → θ(throw loc, S, Π, A)

P-Call-2

θ(v.m(v. throw loc e..)], S, Π, A) → θ(throw loc, S, Π, A)

P-Set

θ(throw loc.f = fi, S, Π, A) → θ(throw loc, S, Π, A)

P-Get

θ(throw loc.[f], S, Π, A) → θ(throw loc, S, Π, A)

P-If

θ(if(v){throw loc}else{e2}], S, Π, A) → θ(throw loc, S, Π, A)

P-Define

θ(← var = throw loc; e2], S, Π, A) → θ(throw loc, S, Π, A)

Figure 18: Ptolemy's operational semantics for exception propagation, based on [8]

(Announce)

θ(c event p throws {pvvar contract} ∈ CT
loc ∉ dom(S) H = bindp(p, S, A) ρ = {vari → vi | i ∈ [1, n]}
S' = S ⊕ (loc → εClosure(H, e, ρ)) Π' = Π ⊕ (loc : thunk c p)
θ(announce p (v) {e}, S, Π, A) → θ(loc. invoke [], S', Π', A)

(Invoke)

θ(eClosure(ε, e, ρ) = S(loc)
θ(loc. invoke [], S, Π, A) → θ[ε, S, Π, A]

Figure 19: Rest of Ptolemy's operational semantics, based on [8, 32]
subject and observers and are omitted in this section.

The following definitions are necessary in progress and preservation arguments for the soundness proof.

**Definition 1.** (Location loc has type t in store S) Location loc has type t in store S, written as S(l) : Π(t) where t = Π(l), if one of the following conditions hold:

(I) type t is a class, i.e. isClass(t), and for some class name c with a set of fields F all the following holds:

(a) S(loc) = [c.F] and Π(loc) = t and c :<: t
(b) dom(F) = dom(fieldsOf(c)) and rng(F) ⊆ (dom(S) ∪ {null})
(c) ∀f ∈ dom(F) if F(f) = loc < c and fieldsOf(c)(f) = u and S(loc) = [u.F] then u' :<: u.

(II) type t is an event closure type, i.e. isThunk(t), where t = thunk:t for some class name c, event type p, list of handlers H, environment ρ, expression e and class name c all the following holds:

(a) S(loc) = eClosure(H, e, ρ)
(b) Π(l) : e :<: c and e :<: c
(c) ∀f ∈ dom(contextOf(p)), either ρ(f) = null or p(f) = loc' and S(loc') = [c.F] and ρ' :<: contextOf(p)(f)
(d) ∀h :<: loc', m ≥ H, Π(loc') = c and (c2, m(t1 var1)) = methodBody(c2, m) then t1 = t

**Definition 2.** (Store S is consistent with store typing Π) Store S is consistent with store typing Π and typing context Γ, written Π, if and only if all the following conditions hold:

(a) dom(S) = dom(Π)
(b) ∀l ∈ dom(S), S(l) : Π(l), i.e. S(l) has type Π(l).
\(\Gamma, a : t' | \Pi \vdash e : t \text{ and } a \text{ is not free in } e, \text{ then } \Gamma | \Pi \vdash e : t\)

\(\text{LEMMA 3. (Environment extension)}\)
\[
\Gamma | \Pi \vdash e : t \text{ and } a \in \text{dom}(\Gamma) \text{ then } \Gamma, a : t' | \Pi \vdash e : t
\]

\(\text{LEMMA 4. (Replacement)}\)
\[
\Gamma | \Pi \vdash \exists e : t \text{ and } \Gamma | \Pi \vdash e : t' \text{ and } \Gamma | \Pi \vdash e' : t' \text{ then } \Gamma | \Pi \vdash e' : t
\]

\(\text{LEMMA 5. (Replacement with subtyping)}\)
\[
\Gamma | \Pi \vdash e : t \text{ and } \Gamma | \Pi \vdash e : u \text{ and } \Gamma | \Pi \vdash e' : u' \text{ such that } u' < u \text{ then } \Gamma | \Pi \vdash e' : t' \text{ where } t' < t
\]

\(\text{B.1 Progress}\)

\(\text{THEOREM 1. (Progress)}\)
\[
\text{Let } (c, S, \Pi, A) \text{ be a configuration with a well typed expression } c, \text{ store } S, \text{ store type } \Pi \text{ and active object list } A, \text{ such that store } S \text{ is consistent with store type } \Pi, \text{ i.e. } \Gamma | \Pi \vdash S \subseteq S. \text{ If } e \text{ has type } t, \text{ i.e. } \Gamma | \Pi \vdash e : t \text{ then either}
\]

- \(e = \text{loc or } e = \text{raise loc and } \text{loc} \in \text{dom}(S)\)
- \(e = \text{null or}\)
- \(\text{one of the following holds:}\)
  - \(\langle e, S', \Pi', A' \rangle\)
  - \(\langle e, S, \Pi', A' \rangle\)

\(\text{Proof Sketch:}\)
\[
\text{Proof is by cases on evaluation of expression } e:
\]

1. \(e = \text{raise loc}. \text{ Since } e \text{ is well-typed and using (T-Raise), } \text{loc} \in \text{dom}(\Pi). \text{ Using store consistency } \Gamma | \Pi \vdash S, \text{loc} \in \text{dom}(S).\)
2. \(e = \text{loc}. \text{ A similar argument applies here where (T-Loc) is used instead of (T-Raise).}\)
3. \(e = \text{null}. \text{ The case is trivial.}\)

Other rules regarding expressions which throw and handle exceptions \[8\] and exception-aware specification expressions are:

4. \(e = \text{throw loc}. \text{ Using (T-Throw) and store-consistency and well-typedness of loc, loc} \in \text{dom}(S) \text{ which in turn allows (T-Throw) to take an evaluation step.}\)
5. \(e = \text{try } \{ \text{raise loc } \} \text{catch(x var)[e]} \). Using (T-Catch), (T-Raise) and store-consistency, \(\text{loc} \in \text{dom}(S) \text{ and } [c.F] = S(\text{loc})\). This allows (R-Catch) to take an evaluation step if \(c < x\). Otherwise, rule (X-Catch) takes an step.
6. \(e = \text{try } \{ v \} \text{catch(x var)[e]} \). Using (T-Catch) which ensures well-typedness of body of try and its catch clauses, (Try-Catch) can take an evaluation step.
7. \(e = \text{if } \{ \text{exits } e \} \text{then } \{ \text{catch(x var)[e]} \} \text{if } \{ \text{exits } e \} \). Using (T-Ref) which ensures precondition is well-typed which in turn allows (Refining) to take a step and reduce to an \text{evalpostexc} expression, if the precondition holds, i.e. \(n \neq 0\). Otherwise, (X-Ref) takes a step. Rule (T-Refining) also ensures well-typedness of its postcondition \(e\) and body.
e. Well-typedness of postcondition allows (EvalPostExc-1) to take an evaluation step, if the postcondition holds. In case of violation of postcondition (X-EvalPostExc-1) takes a step. Well-typedness of body of the refining expression ensures \( \text{loc} \in \text{dom}(S) \) and thus \( [e,F] = S(\text{loc}) \) which allows (EvalPostExc-2) to take a step, if any unchecked exception thrown by the body is in a \( \subseteq \) relation with allowed exception set \( \mathcal{E} \). If the thrown exception is not an allowed exception, then (X-EvalPostExc-2) takes an step.

The following rules for announcement and handling of events as well as registration of observers are based on Ptolemy [22].

8. \( e = \mathbb{E}[\text{announce } p(\mathcal{T})] \). Using well-typedness of \( e \) and (T-Announce), event type \( p \) is a declared event type in class table \( CT \). (T-Announce) ensures all the context variables of \( p \) are passed to the announce expression with appropriate types which in turn allows (Announce) to construct the event closure and take an evaluation step.

9. \( e = \mathbb{E}[\text{loc.invoke}()] \). Using (T-Invoke) and store-consistency, \( \text{loc} \in \text{dom}(S) \) and \( \Pi(\text{loc}) = \text{thunk} p \) which ensures \( \text{loc} \) is pointing to an event closure in the store for event \( p \). If list of observer handlers \( H \) is not empty, then based on part (d) of Definition \( \Pi \) location \( \text{loc} \), pointing to the first observer handler in the event closure, is well-typed and thus \( \text{loc} \in \text{dom}(S) \) which allows (Invoke) to take an step. Otherwise, if \( H \) is empty, (InvokeDone) takes an step.

10. \( e = \mathbb{E}[\text{register loc}()] \). Using (T-Register) and store-consistency, (Register) can take a step by adding a well-type location \( \text{loc} \) to the list of active objects \( A \).

11. \( e = \mathbb{E}[\text{register null}] \) is trivial.

12. \( e = \mathbb{E}[\text{throw loc.invoke}()], e = \mathbb{E}[\text{register throw loc}], e = \mathbb{E}[\text{announce } p(\mathcal{T}.\text{throw loc } e\ldots)] \) where \( e' = \mathbb{E}[\text{throw loc}] \) are all trivial too.

And finally the following cases for standard object-oriented (OO) expressions either could be easily adapted from MiniMAO or are trivial.

13. \( e = \mathbb{E}[\text{loc}.f], e = \mathbb{E}[\text{loc}.f = v], e = \mathbb{E}[\text{cast } t \text{ loc}, e = \mathbb{E}[\text{loc}.m(\mathcal{T})] \).

14. \( e = \mathbb{E}[t \text{ var } = v; e], e = \mathbb{E}[\text{if}(v){e_1} \text{ else }{e_2}], e = \mathbb{E}[\text{new } c()] \) are trivial.

15. \( e = \mathbb{E}[\text{null.m}(\mathcal{T})], e = \mathbb{E}[\text{null}.f], e = \mathbb{E}[\text{null}.f = v], e = \mathbb{E}[\text{cast } c\text{ null}] \) are also trivial.

16. \( e = \mathbb{E}[\text{throw loc}.m(\mathcal{T})], e = \mathbb{E}[\text{m}(v)\text{ throw loc } e\ldots] \), \( e = \mathbb{E}[\text{throw loc}.f = e], e = \mathbb{E}[\text{if}(v)\text{ throw loc } e_2], e = \mathbb{E}[t \text{ var } = \text{throw loc } e_2] \) where \( e' = \mathbb{E}[\text{throw loc}] \) are all trivial too.

B.2 Preservation

Theorem 2. (Preservation)

Let \( e \) be an expression, \( S \) a store, \( \Pi \) a store typing and \( A \) a list of active objects where \( S \) is consistent with store typing \( \Pi \), i.e. \( \Pi \cap S \neq \emptyset. \) If \( \Pi \cap S \vdash e : t \) and \( (e,S,\Pi,A) \rightarrow (e',S',\Pi',A') \) then \( \Pi' \cap S' \neq \emptyset \) and there exists a \( t' \) such that \( t' \leq t \) and \( \Pi \cap S \vdash e' : t' \).

In the above definition \( \Pi' \) is the store typing built and maintained in Ptolemy’s operational semantics.

Proof Sketch:

The proof is by cases on the evaluation relation \( \rightarrow \) :

1. (Throw). \( e = \mathbb{E}[\text{throw loc}], e' = \mathbb{E}[\text{raise loc}] \).

Store consistency is trivial since neither store nor store typing changes.

Now we show \( \Pi \cap S \vdash \mathbb{E}[\text{raise loc}] : t' \) for some \( t' < t \). Using (T-Raise), \( \Pi \cap S \vdash \text{raise loc} : \bot \) and \( \bot \leq t \) for all \( t \), i.e. the bottom type \( \bot \) is subtype of any type.

2. (R-TryCatch). \( e = \mathbb{E}[\text{try} \{\text{raise loc}\} \text{ catch}(x \text{ var})\{e\}], e' = \mathbb{E}[\text{loc}/\text{var}] e \) where \( [e,F] = S(\text{loc}) \), \( e < x \).

Store consistency is trivial.

Now we show \( \Pi \cap S \vdash \mathbb{E}[\text{loc}/\text{var}] e : t' \) for some \( t' < t \). Let \( \Pi \cap S \vdash \text{try} \{\text{raise loc}\} \text{ catch}(x \text{ var})\{e\} : u \). Using (T-TryCatch), \( \Pi \cap S \vdash e : u \) and using Lemma 1 \( \Pi \cap S \vdash \text{loc}/\text{var}\{e\} : u' \) such that \( u' < u \). Finally using Lemma 4 \( t' < t \).

3. (TryCatch). \( e = \mathbb{E}[\text{try} \{e\} \text{ catch}(x \text{ var})\{e\}], e' = \mathbb{E}[v] \).

Store consistency is trivial.

Now we show \( \Pi \cap S \vdash \mathbb{E}[v] : t' \) for some \( t' < t \). Let \( \Pi \cap S \vdash \text{try} \{e\} \text{ catch}(x \text{ var})\{e\} : u \). Using (T-TryCatch), \( \Pi \cap S \vdash v : u \). Using Lemma 5 and reflexivity of subtyping relation \( \subseteq \) we have, \( t' < t \).

4. (Refining).

\( e = \mathbb{E}[\text{refining requires } n \text{ ensures } ev \text{ throws } \mathcal{E}\{e\}], e' = \mathbb{E}[\text{evalpostexc } ev \text{ } ev \mathcal{T} n \mathcal{E}\{e\}] \) where \( n \neq 0 \).

Store consistency is trivial again.

Now we show \( \Pi \cap S \vdash \mathbb{E}[\text{evalpostexc } ev \text{ } ev \mathcal{T} n \mathcal{E}\{e\}] : t' \) for some \( t' < t \). Let \( \Pi \cap S \vdash \text{refining requires } n \text{ ensures } ev \text{ throws } \mathcal{E}\{e\} : u \). Using (T-Refining), \( \Pi \cap S \vdash e : u \). Using (T-EvalPostExc) \( \Pi \cap S \vdash \text{evalpostexc } ev \text{ } ev \mathcal{T} n \mathcal{E}\{e\} : u \). Using Lemma 5 and reflexivity of subtyping relation we have \( t' < t \).

5. (EvalPostExc-1). \( e = \mathbb{E}[\text{evalpostexc } v \text{ } v \mathcal{T} n \mathcal{E}\{e\}], e' = \mathbb{E}[v] \) where \( n \neq 0 \).

Store consistency is trivial since neither store nor store typing changes.

Now we show \( \Pi \cap S \vdash \mathbb{E}[v] : t' \) for some \( t' < t \). Let \( \Pi \cap S \vdash v : u \). Using (T-EvalPostExc), \( \mathbb{E}[\text{evalpostexc } v \text{ } v \mathcal{T} n \mathcal{E}\{e\}] : u \). Using Lemma 5 and reflexivity of subtyping relation we have \( t' < t \).

6. (EvalPostExc-2). \( e = \mathbb{E}[\text{evalpostexc } \text{raise loc} \text{ } n \mathcal{E}\{e\}], e' = \mathbb{E}[\text{raise loc}] \) where \( [e,F] = S(\text{loc}) \) and \( \{e\} \subseteq \mathcal{E} \).

Store consistency is trivial.

Now we show \( \Pi \cap S \vdash \mathbb{E}[\text{raise loc}] : t' \) for some \( t' < t \). Using (T-Raise), \( \Pi \cap S \vdash \text{raise loc} : \bot \) and \( \bot \leq t \) for all \( t \).

Expressions which may throw exceptions are the followings. Ptolemy’s treatment of exceptions is different compared to MiniMAO and Ptolemy where throwing of an exceptions in those languages causes the program to halt.

7. (X-Refining).

\( e = \mathbb{E}[\text{refining requires } n \text{ ensures } ev \text{ throws } \mathcal{E}\{e\}], e' = \mathbb{E}[\text{raise loc}] \) where \( n = 0 \).

Here \( e \) is reduced to an error condition \( \downarrow \) which is not applicable to subject reduction theorem.
8. (X-EvalPostExc-1), (X-EvalPostExc-2). The same argument above, for (X-Refining), applies to these rules too.

9. (X-Throw).
\[ e = \texttt{E[throw null]}, \quad e' = \texttt{E[throw new NullPointerEception()]} \]

Store consistency is trivial.

Now we show \( \Pi \vdash e \) for some \( t' \) for some \( t' < t \). Using (T-Throw), \( \Pi \vdash \text{throw new NullPointerEception()} : \bot \) and \( \bot \vdash e < t \) for all \( t \).

10. (X-TryCatch).
\[ e = \texttt{E[try{[raise loc]} catch{x var}{[e]}]}, \quad e' = \texttt{E[raise loc]} \]

where \([c.F] = S(\text{loc})\) and \( e < x \).

Store consistency is trivial again.

Now we show \( \Pi \vdash \text{raise loc} \) for some \( t' \) for some \( t' < t \). Using (T-Raise), \( \Pi \vdash \text{raise loc} : \bot \) and \( \bot \vdash e < t \) for all \( t \).

11. (X-Register).
\[ e = \texttt{E[register null]}, \quad e' = \texttt{E[throw new NullPointerEception()]} \]

Store consistency is trivial.

Now we show \( \Pi \vdash \text{throw new NullPointerEception()} \) for some \( t' < t \). Using (T-Throw), \( \Pi \vdash \text{throw new NullPointerEception()} : \bot \) and \( \bot \vdash e < t \) for all \( t \).

12. (X-Set), (X-Get), (X-Call), (X-Cast). The same argument above, for (X-Register), applies to these rules too.

Proofs for expressions which announce and handle events as well as register observers are based on the proofs in Ptolemy [32]. However, Ptolemy uses a stack based operational semantics compared to substitution based operational semantics of Ptolemy, which requires such proof rules to be modified. Proofs for such rules are as the following:

13. (Announce).
\[ e = \texttt{E[announce p \{\tau\} \{t\}]}, \quad e' = \texttt{E[loc.invoke()]}, \quad \text{where } (e \text{ event } p \{\tau | \text{var} | \text{contract}\} \in CT, \text{ loc } \notin \text{ dom}(S), H = hbind(p, S, A), p = \{\text{var}, \rightarrow v_i | i \in [1, n]\}, S' = S \oplus (\text{loc } \rightarrow \text{eClosure}(H, e, p)), \text{ and } \Pi' = \Pi \oplus (\text{loc } \oplus \text{thunk } c p) \]

To show store consistency, \( \Pi' \preceq S' \), part (a) of Definition 2 holds since (Announce) adds a fresh location to the domain of both store and store typing. Part (b) of store consistency definition holds for all locations \( \text{loc } \neq \text{loc} \), according to the domain of both store and store typing. To show that part (b) holds for loc, we have to show part (II) of Definition 2 holds for loc.

Part (a), of part (II) of Definition 2 holds, since \( S'(\text{loc}) = \text{eClosure}(H, e, p) \) and \( \Pi'(\text{loc}) = \text{thunk } c p \). Part (b) holds since using (T-Announce), if \( \Pi' \vdash e > c \) then \( e' < c \). For part (c) for all \( f \in \text{dom} \text{contextOf}(p) \), \( \rho(f) = \text{null} \) or \( \rho(f) = \text{loc} \). Part (c) holds trivially if \( \rho(f) = \text{null} \). Otherwise \( \rho(f) = S(\text{loc}) \). If \( e' = S(\text{loc}) \) then \( \Pi' \vdash \text{loc } \oplus \text{loc } \oplus c' \) and (T-Announce) ensures \( c' < \text{contextOf}(f) \). Then sing Lemma 3, we have \( \Pi' \vdash \text{loc } \oplus c' \) where \( c' < \text{contextOf}(f) \).

Now we show \( \Pi' \vdash [\text{loc.invoke()}] : t' \) for some \( t' \) t'. Let \( \Pi' \vdash \text{announce p}[\tau]\{t\} : t \). Using (T-Announce), \( t \text{ event } p\{\ldots\} \in CT \) and if \( \Pi' \vdash e > u \) then \( u < t \).

14. (InvokeDone).
\[ e = [\text{loc.invoke()}], \quad e' = \texttt{E[e']} \]

Store consistency is trivial again.

Now we show \( \Pi' \vdash [\text{loc.invoke()}] : t' \) for some \( t' \). Let \( \Pi' \vdash \text{loc } \cdot u \) and \( \Pi' \vdash \text{loc.invoke()} \cdot u \). Using (T-Invoke), \( \Pi' \vdash \text{thunk } u p \cdot f \). Using store consistency and Definition II part (II) item (b) and assumption \( \text{eClosure}(e, e', \rho) = S(\text{loc}) \), we have \( u' \cdot u \).

Finally using Lemma 4, \( t' < t \).

15. (Invoke).
\[ e = \texttt{E[loc.invoke()]} \]

where \( \text{eClosure}(e, e', \rho) = S(\text{loc}) \) and \( \rho(f) = \text{null} \). Using (T-Invoke), \( \Pi' \vdash \text{loc.invoke()} : t' \). Let \( \Pi' \vdash \text{loc.invoke()} \cdot u \). Using store consistency and Definition II part (II), \( u' < u \). And using Lemma 5, \( t' < t \).

16. (ECGet).
\[ e = \texttt{E[loc.f]}, \quad e' = \texttt{E[v]} \]

where \( \text{eClosure}(H, e', \rho) = S(\text{loc}) \) and \( v = \rho(f) \).

Showing store consistency is trivial.

Now we show \( \Pi' \vdash e[v] : t' \) for some \( t' < t \). Let \( \Pi' \vdash \text{loc.f} : u \) and \( \Pi' \vdash u : u' \). Using store consistency and part(c) of Definition II part (II), \( u' < u \). And using Lemma 5, \( t' < t \).

17. (Register).
\[ e = \texttt{E[register(loc)]}, \quad e' = \texttt{E[loc]} \]
Store consistency is trivial.

Now we show \( \Gamma \Pi \vdash E[\text{loc}] : t' \) for some \( t' < t \). Let \( \Gamma \Pi \vdash \text{register}(\text{loc}) : u \) and \( \Gamma \Pi \vdash \text{loc} : u' \). Using \( (T\text{-REGISTER}) \), \( u' = u \). Using Lemma 5, we have \( \Gamma \Pi \vdash E[\text{loc}] : t' \) for some \( t' < t \). Note that subtyping relation \( < \) is a reflexive transitive relation.

18. \((P\text{-INVOKE}), (P\text{-REGISTER}), (P\text{-ANNOUNCE})\). For \( e = E[\text{throw loc}.\text{invoke}();] \) or \( e = E[\text{register} \text{throw} \text{loc}] \) or \( e = E[\text{announce} p(v, \text{throw loc} e..)] \) where \( e' = E[\text{throw} \text{loc}] \).

Store consistency is trivial.

Now we have to show \( E[\text{throw} \text{loc}] : t' \) for some \( t' < t \). Using \( (T\text{-RAISE}) \) we have \( \Gamma \Pi \vdash \text{throw} \text{loc} : \bot \) which is subtype of any other type \( t \).

And finally proofs for standard object-oriented (OO) expressions are as the following:

18. Proofs for standard OO expressions as in rules \((\text{NEW}), (\set{}), (\text{GET}), (\text{CAST}), (\text{NCAST})\) and \((\text{CALL})\) could be easily constructed by adapting MiniMAO proofs for the same rules.

19. \((P\text{-CALL}-1), (P\text{-CALL}-2), (P\text{-SET}), (P\text{-IF}), (P\text{-DEFINE})\). For \( e = E[\text{throw} \text{loc}.m(\pi)'] \) or \( e = E[p.m(v \text{\text{...throw loc} e..}')] \), \( e = E[\text{throw} \text{loc}.f = e] \) or \( e = E[p.m(v \text{\text{..throw loc} e..}')] \) or \( e = E[t \text{\var} = \text{throw loc} e..] \) where \( e' = E[\text{throw} \text{loc}] \).

Store consistency is trivial.

Now we have to show \( E[\text{throw} \text{loc}] : t' \) for some \( t' < t \). Using \( (T\text{-RAISE}) \) we have \( \Gamma \Pi \vdash \text{throw} \text{loc} : \bot \) which is subtype of any other type \( t \).

C. REFERENCES


[34] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. TOSEM’03, 12(2).

specified by model programs. In *OOPSLA*’07.


[40] M. van Dooren and E. Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *OOPSLA*’05.
