On Exceptions, Events and Observer Chains

Mehdi Bagherzadeh Hridesh Rajan Ali Darvish
Dept. of Computer Science, Iowa State University
Ames, IA, 50011, USA
{mbagherz,hridesh,ali2}@iastate.edu

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 challenge in Ptolemy and propose *Ptolemy*$_{\mathcal{X}}$ that enables modular reasoning about behaviors and flows of exceptions for event announcement and handling. *Ptolemy*$_{\mathcal{X}}$'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. Greybox specifications specify the flows of these exceptions among the observers in the chain. *Ptolemy*$_{\mathcal{X}}$'s type system and refinement rules enforce these specifications and thus enable its modular reasoning. We evaluate the utility of *Ptolemy*$_{\mathcal{X}}$'s exception flow reasoning by applying it to understand a set of aspect-oriented (AO) bug patterns. We also present *Ptolemy*$_{\mathcal{X}}$'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 [29]. Understanding exceptions, especially their behaviors and flows may help with their better use and handling [28, 35]. Previous work, such as JML [25], ESC/Java [12], the work of Jacobs *et al.* [16] anchored exceptions [39], Jex [33] and EJFlow [5] 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 [31] 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 [2], join point interfaces (JPI) [36], crosscutting programming interfaces (XPI) and the work of Khatchadourian *et al.* [20], provides the subject with the knowledge about behaviors of its unknown observers via rely-guarantee-like [7] techniques. The rely-guarantee allows the subject to rely on the provided knowledge about its observers independent of the system configuration [19]. However, the previous work is mostly focused on normal behaviors of the observers and is less concerned about their exceptions, and behaviors and flows of these exceptions. Previous work on Join Point Interfaces (JPI) [3] provides the subject with the knowledge about the types of the exceptions of its observers but not their behaviors or flows.

In summary, the following problems exist when understanding a subject in Ptolemy, especially modular reasoning about behaviors and flows of exceptions in announcement and handling of its event:

- *problem (1)*: subject does not know about the exceptions that its observers may throw;
- *problem (2)*: subject does not know about the behaviors of the exceptions of its observers;
- *problem (3)*: subject does not know about the flows of the exceptions among its observers in the chain of the observers.

We propose *Ptolemy*$_\mathcal{X}$, as an exception-aware extension of Ptolemy, to solve these problems. To address the problem *(1)* for checked exceptions, similar to JPI, we limit the set of checked exceptions of a subject and its observers. These exceptions are referred to as *boundary exceptions*. To address the problem *(1)* for unchecked exceptions we propose *exception-aware specification expressions* to limit the set of unchecked exceptions of the subject and its observers. For the problem *(2)* we use exceptional postconditions to specify the state of the subject and observers upon throwing the exceptions. And finally to solve the problem *(3)*, we use greybox specifications to specify the flow of the exceptions among the observers in their chain, by limiting their implementation structures. *Ptolemy*$_\mathcal{X}$'s type system, refinement rules and runtime assertion checks ensure that the subject and its observers satisfy these specifications and thus enable modular reasoning about the behaviors and flows of the exceptions for announcement and handling of the event, independent of system configuration, i.e. individual observers and their execution order in their chain.

1.1 Contributions

In summary the contributions of this paper are the following:

- Enabling modular reasoning about behaviors and flows of exceptions for event announcement and handling in *Ptolemy*$_\mathcal{X}$;
- Evaluating *Ptolemy*$_\mathcal{X}$'s exception flow reasoning in understanding Coelho *et al.* 's aspect-oriented bug patterns [6];
- Presenting *Ptolemy*$_\mathcal{X}$'s sound static and dynamic semantics.

The rest of the paper continues as the following. Section 2 illustrates the problems *(1)–(3)* for modular reasoning about behaviors and flows of exceptions in announcement and handling of events in Ptolemy. Section 3 describes our proposed language design of *Ptolemy*$_\mathcal{X}$ and how it solves the example modular reasoning problems of Section 2. Section 4 discusses *Ptolemy*$_\mathcal{X}$'s sound type system and its refinement rules that enable its modular reasoning. Section 5 illustrates the usability of *Ptolemy*$_\mathcal{X}$'s exception flow reasoning in understanding Coelho *et al.* 's AO bug patterns [6]. This section also discusses the overhead of applying *Ptolemy*$_\mathcal{X}$ to Ptolemy programs. Section 6 compares our proposal with existing work. Section 7 concludes and discusses future work.

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 [31].

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 Φ_b on line 11 of Figure 1. Figure 1 and Figure 2 illustrate a savings bank account with a withdraw functionality. The assertion Φ_b 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 Φ_b one should understand the exceptional behavior 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 [25]. For such understanding to be modular [23], 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 29–38 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 *(1)*, 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 II extension of Java with support for explicit announcement and handling of typed events [31]. 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 26. 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 [21, 22], these observers form a chain

```
1  class Savings {
2   int bal; int numWithdrawals;
3   void withdraw(int amt) throws Throwable{
4    try{
5     announce WithdrawEv(this,amt){
6       bal-= amt;
7       numWithdrawals++;
8     }
9    }
10   catch(RegDExc e){
11    //@ assert this.bal==old(this.bal);
12    throw e; }
13   catch(RuntimeException e){ e.printStackTrace(); }
14  } .. }
```

```
15 class Check {
16  void check(WithdrawEv next) throws Throwable{
17   refining
18    establishes next.acc().bal==old(next.acc().bal){
19    if(next.acc().numWithdrawals>=6)
20     throw new RegDExc();
21   }
22   next.invoke();
23  }
24  when WithdrawEv do check;
25
26  Check(){ register(this);  }
27 }
28 class RegDExc extends Exception {}
```

Figure 1: Savings bank account example in Ptolemy [31] with the subject `Savings` and the observer `Check`.

based on their dynamic registration order, with the event body at the end of the chain. Upon announcement of the event the control is transferred from the subject to the first observer in the chain that at some point during its execution may decide to invoke the next observer in the chain, using the **`invoke`** expression, line 22. The **`invoke`** expression is the equivalent of AspectJ's **`proceed`**. The chain of the observers for `WithdrawEv` is stored in an event closure **`next`**, that is passed as a parameter to its observer handler method `check`, line 16. The subject `Savings` does not statically know about its observers or their execution order in the chain.

The event `WithdrawEv` should be declared before it is announced or handled. The declaration of `WithdrawEv` in Figure 2 lists a set of context variables, lines 30–31, and has a greybox translucid contract, lines 32–37. The context variables are shared information between the subject and its observers and their values are provided by the subject when the event is announced, line 5. Translucid contracts [2] are discussed later in this section. The event `WithdrawEv` does not give the subject `Savings` any information about the exceptions of its observers, i.e. problem *(1)*.

```
29 void event WithdrawEv{
30  Savings acc;
31  int amt;
32  requires acc!=null
33  assumes{
34   establishes next.acc().bal==old(next.acc().bal);
35   next.invoke();
36  }
37  ensures acc.bal<=old(acc.bal)
38 }
```

Figure 2: Event `WithdrawEv` with a translucid contract, lines 32–37.

Exception Handling in Ptolemy To cope with the unknown exceptions of its observers, and especially their checked exceptions, the subject `Savings` adds a general throws clause **`throws Throwable`** to the signature of the method `withdraw` which announces `WithdrawEv`, line 3. The same is true for the observer `Check` and its observer handler method `check`, line 16, since it invokes other unknown observers of `WithdrawEv` with their unknown exceptions using the invoke expression. However, these general catch clauses hardly provide any information about the exceptions of the observers of `WithdrawEv` since they basically allow any checked exceptions. As in Java, in Ptolemy checked exceptions are propagated explicitly by declaring them in method headers but unchecked exceptions are propagated implicitly.

Translucid Contracts in Ptolemy The translucid contract of `WithdrawEv`, lines 32–37 of Figure 2, is a greybox specification [4] that enables modular reasoning about the *normal* behavior and control effects of its announcement and handling. The normal behavior of the announcement and handling of `WithdrawEv` is specified using the precondition **`requires`**, line 32, and the postcondition **`ensures`**, line 37. The control effects for the normal behavior are specified by the **`assumes`** block, lines 33–36, that limits the implementation structure of the observers of the event. The assumes block is a combination of program and specification expressions. The program expression, line 35, exposes the control effects of interest in the implementations of the observers such as `Check`, whereas the specification expression, line 34, hides the rest of the implementations of the observers and allows them to vary as long as they respect the specification. The assumes block on lines 33–36 says that observers of `WithdrawEv` can do anything as long as they do not change the balance of the account `acc`, i.e. **`establishes`** **`next`**$.acc().bal ==$ **`old`**(**`next`**$.acc().bal$), and then invoke the next observer in the chain using **`next.invoke`**(). The expression **`next`**$.acc()$ returns the context variable `acc`.

Refinement of Translucid Contracts Refinement rules for the translucid contract of `WithdrawEv` enable modular reasoning about normal behavior and control effects of its announcement and handling, independent of its unknown observers and their execution order [2]. The refinement rules require each subject and observer of `WithdrawEv` to satisfy the pre- and postconditions of its translucid contract. They also require the observers to structurally refine the assumes block of the translucid contract. For the observer handler method `check`, lines 16–23 in Figure 1, to structurally refine the assume block of `WithdrawEv`, lines 33–36 in Figure 2, the following conditions should hold: for each program expression in the assumes block, line 35, the refining observer `check` must have a textually matching program expression in its implementation at the same place, line 22; and for each specification expression in the assumes block, line 34, the observer must have a **`refining`** expression with the same specification in its implementation at the same place, lines 17–21. Using the contract of `WithdrawEv` one may conclude that upon announcement and normal termination of `WithdrawEv` all of its observers in the chain are invoked. This is valid since each observer of `WithdrawEv` has to refine its contract and have the invoke expression in its implementation. However the translucid contract only works for normal execution of announcement and handling of `WithdrawEv`, i.e. no exceptions thrown, and doe not provide any information about the exceptions of its observers, their behaviors or flows, i.e. problems *(1)*–*(3)*.

2.1.2 Boundary Exceptions in Ptolemy

As illustrated, in Ptolemy the subject `Savings` does not know about exceptions of its observers, i.e. problem *(1)*, and the throws clauses **throws Throwable** in the signature of the observer handler methods do not address the problem. Previous work on join point interfaces (JPI) [3] 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.

```
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 Φ_{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.

```
1  class Logger{
2   void log(WithdrawEv next) throws RegDExc{
3    try{
4     refining
5      establishes next.acc().bal==old(next.acc().bal){
6       Log.logTrans(next.acc(),"withdraw");
7     }
8     next.invoke();
9    }
10   catch(Exception e){} /* swallow */
11  }
12  when WithdrawEv do log;
13 }
```

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 of the observers. 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 3 and 10, catches the exception and swallows it, line 10.

To reason about and verify Φ_{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 Φ_{f1}. If `Logger` runs before `Check` then Φ_{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 Φ_{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 Φ_{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 Φ_{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 3 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$_X$*

In this section we describe *Ptolemy$_X$*'s core syntax and specification features for stating behaviors and flows of exceptions during event announcement and handling and address the problems *(1)–(3)*.

3.1 Program Syntax

Ptolemy$_X$ is an exception-aware extension of Ptolemy [31] with support for specification and modular reasoning about behaviors and flows of exceptions during event announcement and handling. Similar to Ptolemy, *Ptolemy$_X$* is an implicit invocation (II) object-oriented (OO) language with support for explicit announcement and handling of typed events. The formal definition of *Ptolemy$_X$* is given as an expression language.

[1]Translucid contracts in Ptolemy [2] do not support throwing and handling of exceptions and only work for normal execution. One way of treating try-catch expressions in observers such as `Logger` is to discard the catch part and only keep the body of the try part, which represents the normal execution. This allows `Logger` to structurally refine its contract in Figure 2.

The syntax of $\mathit{Ptolemy}_{\mathcal{X}}$'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 [8] and are divided into checked and unchecked exceptions. For simplicity, $\mathit{Ptolemy}_{\mathcal{X}}$ does not have packages, privacy modifiers, abstract classes and methods, static members, interfaces or constructors. The superscript $\overline{term}$ shows a sequence of zero or more *term* whereas [*term*] means zero or one, i.e. optional. A $\mathit{Ptolemy}_{\mathcal{X}}$'s program (*prog*) is a collection of declarations ($\overline{decl}$) followed by an expression (*e*), which is like a main method in Java. There are two types of declarations in $\mathit{Ptolemy}_{\mathcal{X}}$: class and event type declarations.

$$
\begin{array}{lll}
prog & ::= & \overline{decl}\; e \\
decl & ::= & \mathbf{class}\; c\; \mathbf{extends}\; d\; \{\; \overline{form}\;\; \overline{meth}\;\; \overline{binding}\; \} \\
 & \mid & c\; \mathbf{event}\; p\; \mathbf{throws}\; \overline{x}\; \{\; \overline{form}\; [\; contract\;]\; \} \\
t & ::= & c \mid \mathbf{thunk}\; c\; p \\
meth & ::= & t\; m\; (\overline{form})\; \{\; e\; \}\; \mathbf{throws}\; \overline{x} \\
form & ::= & t\; var, \quad \text{where } var \neq \mathbf{this} \\
binding & ::= & \mathbf{when}\; p\; \mathbf{do}\; m \\
ep & ::= & var \mid ep.f \mid ep == ep \mid ep < ep \mid !\; ep \mid ep \;\&\&\; ep \mid \mathbf{old}(ep) \\
e, se & ::= & n \mid \mathbf{null} \mid var \mid \mathbf{new}\; c() \mid e.m(\overline{e}) \mid e.f \mid e.f = e \\
 & \mid & \mathbf{if}\; (ep)\; \{\; e\; \}\; \mathbf{else}\; \{\; e\; \} \mid \mathbf{cast}\; c\; e \mid form = e;\; e \\
 & \mid & \mathbf{announce}\; p\; (\overline{e})\; \{\; e\; \} \mid e.\mathbf{invoke}() \mid \mathbf{register}(e) \\
 & \mid & \mathbf{refining}\; spec\; \{\; e\; \} \mid spec \mid \mathbf{either}\; \{\; e\; \}\; \mathbf{or}\; \{\; e\; \} \\
 & \mid & \mathbf{throw}\; e \mid \mathbf{try}\; \{e\}\; \mathbf{catch}\; (x\; var)\; \{e\} \\
contract & ::= & \mathbf{requires}\; ep\; [\; \mathbf{assumes}\; \{\; se\; \}\;]\; \mathbf{ensures}\; ep\; \overline{excp} \\
excp & ::= & \mathbf{signals}\; (\overline{x})\; ep \\
spec & ::= & \mathbf{requires}\; ep\; \mathbf{ensures}\; ep\; \mathbf{throws}\; \overline{x}
\end{array}
$$

where

$$
\begin{array}{rcl}
c & \in & \mathcal{C}, \text{ set of class names} \\
d & \in & \mathcal{C} \cup \{\mathbf{Object}\}, \text{ a set of superclass names} \\
p & \in & \mathcal{P}, \text{ set of event type names} \\
f & \in & \mathcal{F}, \text{ set of field names} \\
var & \in & \{\mathbf{this}, \mathbf{next}\} \cup \mathcal{V}, \mathcal{V} \text{ is a set of variable names} \\
x & \in & \mathcal{X} \cup \{\mathbf{NoRuntimeExc}, \mathbf{Exception}, \mathbf{RuntimeException} \\
 & & \mathbf{ClassCastException}, \mathbf{NullPointerException}\}, \\
 & & \mathcal{X} \text{is a set of exception names}
\end{array}
$$

Figure 5: $\mathit{Ptolemy}_{\mathcal{X}}$'s syntax, based on [2, 31]

3.1.1 Declarations

In a class declaration, the class has a name (*c*), a super class (*d*), a set of fields ($\overline{form}$), methods ($\overline{meth}$) and binding declarations ($\overline{binding}$). A binding declaration associates an event type (*p*) with an observer handler method (*m*). In $\mathit{Ptolemy}_{\mathcal{X}}$ 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 ($\overline{x}$) 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 (*c*), a set of checked boundary exceptions ($\overline{x}$), a set of context variables ($\overline{form}$) 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 *(1)* for checked exceptions. For example, Figure 3 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.

3.1.2 Specifications

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** $(\overline{x})$ *ep* lists checked or unchecked exceptions $(\overline{x})$ of the subjects and observers 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 $\overline{x}$ then it terminates in a state that satisfies the exceptional postcondition *ep*. In other words, if an observer or a subject throws an exception $(\overline{x})$ then it must satisfy the exceptional postcondition (*ep*). Postcondition 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 postconditions 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 [2]. The normal behavior **requires** ep_1 **ensures** ep_2 says that if the event is announced in a state that satisfies the precondition ep_1 and its announcement and handling terminates normally, i.e. throws no exceptions, then it terminates in a state that satisfies the postcondition ep_2. The specification **establishes** *ep* is sugar for **requires** $true$ **ensures** *ep*.

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 2–9 with the program expressions on lines 3, 6 and 7 and the exception-aware specification expressions on lines 4–5 and 8.

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** ep_1 **ensures** ep_2 **throws** $\overline{x}$ says that in a refining implementation of an observer, if the execution starts in a state satisfying the precondition ep_1 and terminates normally, it terminates in a state satisfying the postcondition ep_2. However, if the execution terminates abnormally by throwing an unchecked exception, then the thrown exception must an exception in $\overline{x}$ otherwise the specification expression is violated. The exception-aware specification expression **requires** ep_1 **ensures** ep_2 is sugar for **requires** ep_1 **ensures** ep_2 **throws RuntimeException**. Exception-aware specification expressions address the problem *(1)* for unchecked exceptions.

For example, Figure 6 illustrates an exception-aware specification expression on lines 4–5 that allows all unchecked exceptions `RuntimeException` 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 3–4, that allows no unchecked exceptions, i.e. **NoRuntimeExc**. The program expressions **next**.**invoke**() in this figure, line 5, cannot throw any unchecked exceptions since it invokes the next observer. Finally, Figure 12 illustrates two exception-aware specification expressions, lines 4–5 and 8. 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 6 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 6. 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 12 an observer can catch an unchecked exception thrown by another observer in the chain and swallow it because its invoke expression, line 6, is enclosed by a try-catch expression, lines 3 and 7, however, it cannot catch checked exceptions such as `RegDExc`. Because invoke expressions plays 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. $Ptolemy_X$'s typing and refinement rules, in Section 4, ensure that the subjects and observers satisfy these specifications.

3.1.3 Expressions

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

In $Ptolemy_X$, a subject explicitly announces an event (p) using an **announce** expression with parameters ($\overline{e}$) as its context variables and the event body (e). The announce expression starts the execution of the chain of observers for the event (p). An observer is registered using a **register** expression that evaluates its parameter (e) to an object and puts it into a chain of observers. Expression **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 16 of Figure 1. The type (***thunk*** $c\ p$) ensures the value of the corresponding actual parameter is of type event closure with return type (c) for an event (p). The expression **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 Φ_b in Section 2.1, $Ptolemy_X$ provides exceptional postconditions. Figure 6 illustrates the exceptional postconditions for the checked boundary exception `RegDExc`, line 9, and unchecked exceptions `RuntimeException`, line 10, The exceptional postcondition for `RegDExc` says that if during announcement and handling of `WithdrawEv` an exception `RegDExc` is thrown by its observers then they must guarantee that the balance of the account `acc` does not change, i.e. $acc.bal == \texttt{old}(acc.bal)$.

```
1  void event WithdrawEv throws RegDExc{ ..
2   requires acc!=null
3   assumes{
4    establishes next.acc().bal==old(next.acc().bal)
5     throws RuntimeException;
6    next.invoke();
7   }
8   ensures acc.bal<=old(acc.bal)
9   signals (RegDExc) acc.bal==old(acc.bal)
10  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 Φ_b in a modular fashion using only the implementation of the subject `Savings`, in Figure 1, and the contract of the event `WithdrawEv` especially its exceptional postcondition for `RegDExc`, line 9. Recall that the assertion Φ_b says if an exception `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 1 if an exception `RegDExc` is thrown by an observer and propagated back to the subject. According to the exceptional postcondition of `RegDExc` the observers of `WithdrawEv` ensure that the predicate $acc.bal == \texttt{old}(acc.bal)$ is true upon throwing `RegDExc`. By replacing the variable **this** for `acc` in this exceptional postcondition we get the predicate $\texttt{this}.bal == \texttt{old}(\texttt{this}.bal)$ which is the same as the assertion Φ_b that we wanted to verify. Recall that **this** is passed as the context variable `acc`, line 5 of Figure 1.

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 $Ptolemy_X$'s refinement rules require all the observers and subjects of `WithdrawEv` to satisfy the exceptional postcondition of `RegDExc` if they throw it. The refinement rules are discussed in Section 4.

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 Φ_{f1} and Φ_{f2} in Section 2.2, $Ptolemy_X$ provides exception-aware specification expressions and translucid contracts. The contract in Figure 6 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 Φ_{f1} holds. Recall that Φ_{f1} says if an exception `RegDExc` 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 **`catch`** (**`RuntimeException`** e) $\{..\}$ in `Savings`, line 13 of Figure 1 is necessary, i.e. Φ_{f2}, since an unchecked exception thrown by an observer propagates back to the subject. However, if the contract in Figure 11 is used instead of the contract in Figure 6 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 Φ_{f1} and Φ_{f2}, independent of system configuration, i.e. unknown observers or their execution order in their chain. This is only valid because *Ptolemy*$_{\chi}$'s type system and refinement rules ensure that subjects and observers of an event respect their contract.

4. MODULAR REASONING AND REFINEMENT IN *Ptolemy*$_{\chi}$

Ptolemy$_{\chi}$ 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*$_{\chi}$'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$_{\chi}$'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 [3] 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$_{\chi}$ tying rules use the type attributes of Figure 7 in which regular types are augmented with a set of checked exceptions $\overline{x}$.

The type attribute **`exp`** $t, \overline{x}$ denotes expressions of type t that may throw the checked exceptions $\overline{x}$. Variables do not throw exceptions and are denoted by **`var`** t. The type attribute OK is used to type check top level declarations whereas OK in c is used for type checking of lower level declarations in the context of an upper level declaration c. In *Ptolemy*$_{\chi}$, similar to Java, a checked

$\theta ::=$	"type attributes"
OK	"program/top-level declaration"
$\mid$ OK in c	"method, binding"
$\mid$ **var** t	"var/formal/field"
$\mid$ **exp** $t, \overline{x}$	"expression"
where $\forall x \in \overline{x}.\ isChecked(x)$	
$\mid \perp, \overline{x}$	"bottom type"
$\Gamma ::= \{var : t\}$	"type environment"
where $var \subseteq (\{\textbf{this}, \textbf{next}\} \cup \mathcal{V})$,	
$\Pi ::= \{loc : t\}$	"store typing"
where $loc \subseteq \mathcal{L}$,	
$\mathcal{L}$ is set of location names	
$\Gamma \mid \Pi\ \vdash e : \theta$	"typing judgement"

Figure 7: Type attributes, based on [31].

exception is a subtype of **`Exception`** that is not a subtype of **`RuntimeException`**. The auxiliary function *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 $\Gamma|\Pi\ \vdash e : \theta$ says that in the typing environment Γ and store typing Π the expression e has the type θ. Figure 8 shows select typing rules for *Ptolemy*$_{\chi}$. The full set of typing rule could be found in our technical report [1].

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 [3], the observers can throw any subtype of the boundary exceptions and the subjects can only throw them invariantly. 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_b if a subject throws E_s and an observer throws E_o, such that $E_s <: E_o <: E_b$, then the subject cannot handle E_o [3]. 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) type checks an observer of the event (*p*) and its binding declaration. The rule checks for the relation $\subseteq:$ between the checked exceptions ($\overline{x'}$) of the observer handler method (*m*) of the event (*p*) and its boundary exceptions ($\overline{x}$), i.e. $\overline{x'} \subseteq: \overline{x}$. Similar to non handler methods, the observer handler method (*m*) must declare its checked exceptions ($\overline{x'}$). 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 (*contract*), i.e. $se \sqsubseteq e$. The structural refinement $\sqsubseteq$ is discussed in Section 4.2.

The rule (T-EVENTTYPE) 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 ($\overline{x'}$) of the assumes block (*se*) and the boundary exceptions ($\overline{x}$) of its event (*p*), i.e. $\overline{x'} \subseteq: \overline{x}$. The type (t') of (*se*) should also be a subtype $<:$ of the return type (*c*) 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 invariantly. This means that the set of the checked exceptions ($\overline{x''}$) of the subject must be in a subset relation $\subseteq$ with the boundary exceptions ($\overline{x}$) of the event (*p*), i.e. $\overline{x''} \subseteq \overline{x}$. The relation $\subseteq$ is different from $\subseteq:$ since it does not allow the sub-

(T-BINDING)

$$\frac{\begin{array}{c} t\ m(\texttt{thunk}\ t\ p\ var)\ \texttt{throws}\ \overline{x'}\ \{e\} = CT(c,m) \\ t\ \texttt{event}\ p\ \texttt{throws}\ \overline{x}\ \{\overline{form}\ contract\} \in CT \\ contract = \texttt{requires}\ \ ep_1\ \texttt{assumes}\ \{se\}\ \texttt{ensures}\ \ ep_2\ \overline{excp} \\ \overline{x'} \subseteq: \overline{x} \qquad se \sqsubseteq e \end{array}}{\vdash (\texttt{when}\ p\ \textbf{do}\ m)\ :\ \text{OK in c}}$$

(T-EVENTTYPE)

$$\frac{\begin{array}{c} contract = \texttt{requires}\ \ ep_1\ \texttt{assumes}\ \{se\}\ \texttt{ensures}\ \ ep_2\ \overline{excp} \\ \forall x \in \overline{x}.\ \ isChecked(x) \\ \overline{var : t} \vdash se : \textbf{exp}\ t',\ \overline{x'} \qquad t' <: c \qquad \overline{x'} \subseteq: \overline{x} \end{array}}{\vdash c\ \texttt{event}\ p\ \texttt{throws}\ \overline{x}\ \{\overline{t\ var};\ \ contract\} : \text{OK}}$$

(T-ANNOUNCE)

$$\frac{\begin{array}{c} c\ \texttt{event}\ p\ \texttt{throws}\ \overline{x}\ \{\overline{t'\ var'};\ \ contract\} \in CT \\ \forall e_i \in \overline{e}.\ \Gamma|\Pi\ \vdash e_i : \textbf{exp}\ t_i, \overline{x_i} \qquad \Gamma|\Pi\ \vdash e : \textbf{exp}\ t', \overline{x'} \\ \forall t_i, t_i' \in \overline{t'}.\ \ t_i <: t_i' \qquad t' <: c \qquad \overline{x''} = \bigcup \overline{x_i} \cup \overline{x'} \qquad \overline{x''} \subseteq \overline{x} \end{array}}{\Gamma|\Pi\ \vdash \texttt{announce}\ p\ (\overline{e})\ \{e\} : \textbf{exp}\ c, \overline{x''}}$$

(T-INVOKE)

$$\frac{\begin{array}{c} c\ \texttt{event}\ p\ \texttt{throws}\ \overline{x}\ \{\overline{form}\ \ contract\} \in CT \\ \Gamma|\Pi\ \vdash e : \textbf{exp}\ \texttt{thunk}\ c\ p, \overline{x'} \qquad \overline{x'} == \overline{x} \end{array}}{\Gamma|\Pi\ \vdash e.\texttt{invoke}\ () : \textbf{exp}\ c, \overline{x}}$$

(T-REGISTER)

$$\frac{\Gamma|\Pi\ \vdash e : \textbf{exp}\ t, \overline{x}}{\Gamma|\Pi\ \vdash \texttt{register}\ (e) : \textbf{exp}\ t, \overline{x}}$$

(T-SPEC)

$$\frac{\begin{array}{c} \Gamma|\Pi\ \vdash ep_1 : \textbf{exp}\ t_1, \emptyset \\ \Gamma|\Pi\ \vdash ep_2 : \textbf{exp}\ t_2, \emptyset \qquad \forall\, x \in \overline{x}.\ \ x <: \texttt{RuntimeException} \end{array}}{\Gamma|\Pi\ \vdash \texttt{requires}\ \ ep_1\ \texttt{ensures}\ \ ep_2\ \texttt{throws}\ \overline{x}\ \ : \textbf{exp}\ \bot, \emptyset}$$

(T-REFINING)

$$\frac{\begin{array}{c} spec = \texttt{requires}\ \ ep_1\ \texttt{ensures}\ \ ep_2\ \texttt{throws}\ \overline{x} \\ \Gamma|\Pi\ \vdash spec : \textbf{exp}\bot, \emptyset \qquad \Gamma|\Pi\ \vdash e : \textbf{exp}\ t, \overline{x} \end{array}}{\Gamma|\Pi\ \vdash \texttt{refining}\ \ spec\ \{e\} : \textbf{exp}\ t, \overline{x}}$$

(T-THROW)

$$\frac{\begin{array}{c} \Gamma|\Pi\ \vdash e : \textbf{exp}\ t, \overline{x} \\ t <: \texttt{Throwable} \qquad isChecked(t)\ ?\ \overline{x'} = \{t\} : \overline{x'} = \emptyset \end{array}}{\Gamma|\Pi\ \vdash \texttt{throw}\ e : \textbf{exp}\ \bot, \overline{x'} \cup \overline{x}}$$

(T-TRYCATCH)

$$\frac{\begin{array}{c} \Gamma|\Pi\ \vdash e_1 : \textbf{exp}\ t, \overline{x_1} \\ \Gamma, var : x|\Pi \vdash e_2 : \textbf{exp}\ t, \overline{x_2} \qquad \overline{x} = \{x_i\ | x_i \in \overline{x_1} \wedge\ !(x_i <: x)\} \end{array}}{\Gamma|\Pi\ \vdash \texttt{try}\ \ \{e_1\}\ \texttt{catch}\ \ (x\ var)\ \{e_2\} : \textbf{exp}\ t, \overline{x} \cup \overline{x_2}}$$

Auxiliary Functions:

$isChecked(x) = (x <: \texttt{Exception}) \ \wedge\ !(x <: \texttt{RuntimeException})$

$\overline{x_1} \subseteq: \overline{x_2} = \forall x_1 \in \overline{x_1}, \exists\, x_2 \in \overline{x_2}.\ \ x_1 <: x_2$

Figure 8: *Ptolemy*$_X$'s select typing rules, based on [8, 31].

ject to throw subtypes of the boundary exceptions. The set of the checked exception ($\overline{x''}$) of the subject is the union of the set of checked exceptions ($\overline{x'}$) for the body (e) of its announce expression and the set of checked exceptions ($\overline{x_i}$) for its parameters (e_i), i.e. $\overline{x''} = \bigcup \overline{x_i} \cup \overline{x'}$. 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') of the body (e) of the announce expression and the return type (c) of the event (p).

The rule (T-INVOKE) type 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 (c) and the boundary exceptions ($\overline{x}$) is of type $\texttt{thunk}\ c\ p, \overline{x'}$. The rule checks for the equality of the set checked exceptions ($\overline{x'}$) of the invoke expression and the set of the boundary exceptions ($\overline{x}$) of its event (p), i.e. $\overline{x'}\ ==\ \overline{x}$. The rule (T-REGISTER) says that the set of checked exceptions ($\overline{x}$) of a register expression is set of checked exceptions of its parameter (e).

The rule (T-SPEC) type 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 ($\overline{x}$) 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 $\bot$ that is a subtype of any other type. The rule (T-REFINING) type checks a refining expression that refines an exception-aware specification expression (*spec*). The rule simply says that the set of checked exceptions ($\overline{x}$) of the refining expression is the same as the set of checked exceptions of its body (e).

The rule (T-THROW) type checks a throw expression. It adds the thrown exception (t) to the exception set ($\overline{x}$) of the expression, if the exception is a checked exception. The conditional $isChecked(t)\ ?\ \overline{x'} = \{t\} : \overline{x'} = \emptyset$ 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) type checks a try-catch expression. It computes the set of checked exceptions ($\overline{x_1}$) 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 ($\overline{x}$) 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 ($\overline{x_2}$). The set of the checked exceptions of the try-catch expression is the union of ($\overline{x}$) and ($\overline{x_2}$). The rest of the expression typing rules compose the set of checked exceptions thrown by subexpressions of an expression, based on the compositional rules of the language [1].

4.1.4 Soundness and Dynamic Semantics

Ptolemy$_X$'s type system is proven sound following the 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 *Ptolemy*$_X$'s dynamic and static semantics rules could be found in our report [1].

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 9 shows select rules for *Ptolemy*$_X$'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. $se \sqsubseteq e$, 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 $\texttt{refining}\ spec\{e\}$ at the same place in the implementation which *claims* to refine *spec*. The rule for $\texttt{either}\{se_1\}\texttt{or}\{se_2\}$ 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 $\texttt{try}\ \{se_1\}\ \texttt{catch}\ (x\ var)\ \{se_2\}$ is structurally refined by

c **event** p **throws** $\overline{x}$ $\{\overline{form}\ contract\} \in CT$
$contract =$ **requires** ep_1 **assumes** $\{se\}$ **ensures** ep_2 $\overline{excp}$
when p **do** m and $c\ m($**thunk** $c\ p$ **next**$)\{e\} \in CT$
se is structurally refined by e, $se \sqsubseteq e$, as follows:

Cases of (se)	Refined by (e)	Side conditions
var	var	
$t\ var = se_1; se_2$	$t\ var = e_1; e_2$	$se_1 \sqsubseteq e_1, se_2 \sqsubseteq e_2$
if$(sp)\{se_1\}$ **else**$\{se_2\}$	**if**$(ep)\{e_1\}$ **else**$\{e_2\}$	$sp \sqsubseteq ep$, $se_1 \sqsubseteq e_1, se_2 \sqsubseteq e_2$
either $\{se_1\}$ **or** $\{se_2\}$	e	$se_1 \sqsubseteq e \vee se_2 \sqsubseteq e$
se.**invoke** ()	e.**invoke** ()	$se \sqsubseteq e$
announce $p(\overline{se})$ $\{se\}$	**announce** $p(\overline{e})$ $\{e\}$	$\overline{se} \sqsubseteq \overline{e}$, $se \sqsubseteq e$
try $\{se_1\}$ **catch** $(x\ var)\{se_2\}$	**try** $\{e_1\}$ **catch** $(x\ var)\{e_2\}$	$se_1 \sqsubseteq e_1, se_2 \sqsubseteq e_2$
throw se	**throw** e	$se \sqsubseteq e$
$spec$	**refining** $spec\{e\}$	

Figure 9: Select rules for structural refinement, based on [2,34].

try$\{e_1\}$**catch**$(x\ var)\{e_2\}$ if its subexpressions (se_1) and (se_2) are refined by the subexpressions (e_1) and (e_2).

4.3 Runtime Assertion Checking

Structural refinement rules ensure that for each exception-aware specification expression (*spec*) of the form **requires** ep_1 **ensures** ep_2 **throws** $\overline{x}$ in a translucid contract of an event, there is a refining expression **refining** $spec\{e\}$ which claims to refine (*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 ($\overline{x}$). 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 10 illustrates, in grey, the runtime assertion checking for the observer handler method `check` of Figure 1 with its translucid 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 `RegDExc`, lines 22–25, and `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 `Con.signal` checks for the exceptional postcondition of the exception `e` and aborts the execution or raises an error of type **`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 `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. **`throws RuntimeException`** on line 9. RAC also checks for the pre- and normal postcondition of the observer, lines 4 and 20 and the pre- and normal postcondition of the refining expression, lines 6 and 17, using the methods `Con.require` and `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.

```
1  void check(WithdrawEv next) throws RegDExc{
2   try{
3    /* observer's precondition */
4    Con.require(next.acc()!=null);
5    /* refining precondition */
6    Con.require(true);
7    try{
8     refining establishes next.acc().bal==
9      old(next.acc().bal) throws RuntimeException{
10     if(next.acc().numWithdrawal>=6)
11      throw new RegDExc();
12    }
13   /* allowed unchecked exceptions */
14   } catch(RuntimeException e){
15     Con.allowedExc(e, {RuntimeException.class}); }
16   /* refining postcondition */
17   Con.ensure(next.acc().bal==old(next.acc().bal));
18    next.invoke();
19    /* observer's normal postcondition */
20    Con.ensure(next.acc().bal<=old(next.acc().bal));
21   }
22   catch(RegDExc e){
23    /* exceptional postcondition */
24    Con.signal(next.acc().bal==old(next.acc().bal),e);
25    throw e; }
26   catch(RuntimeException e){
27    /* exceptional postcondition */
28    Con.signal(true, e);
29    throw e; }
30  }
```

Figure 10: Runtime assertion checking in the observer `Check`.

5.1 AO Bug Patterns in *Ptolemy*$_\chi$

Despite their differences [31], *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 **`within`** and **`cflow`** 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 [31].

5.2 Throw Without Catch

In *Ptolemy*$_X$, 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 6 together illustrate a throw without catch bug, if the **`catch`** (**`RuntimeException`** e){..} on line 13 of Figure 1 is inadvertently forgotten by the developer. Similar to Java, *Ptolemy*$_X$ does not complain about the missing catch clause for unchecked exceptions. The contract for `WithdrawEv` in Figure 6 allows the observers of `WithdrawEv` to throw any unchecked exception, lines 4–5. 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 13 of Figure 1 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 6 in which the observers of `WithdrawEv` cannot throw any unchecked exceptions. The exception-aware specification expression on lines 3–4 limits the set of unchecked exceptions of the observers of `Withdraw` to nothing, i.e. **`throws NoRuntimeExc`**.

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

Figure 11: No unchecked exceptions for observers, lines 3–4.

Figure 12 illustrates another variation of the contract for `WithdrawEv` in which each observer has a try-catch expression in its body, lines 3 and 7–8. The try-catch expressions catches, line 7, 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.

```
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 NoRuntimeExc; }
9  }  .. }
```

Figure 12: Observers handle their unchecked exceptions locally, lines 3 and 7 and 8.

Unlike the contract in Figure 11, the contract in Figure 12 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 11.

5.3 Residual Catch

In *Ptolemy*$_X$, 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 contact in Figure 11 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 13. However, the contract for `WithdrawEv` in Figure 11 does not allow its observers to throw any unchecked exceptions, lines 3–4. This in turn means the catch clause on line 13 is not necessary since the subject is trying to catch exceptions that are not thrown by its observers. Recall that in *Ptolemy*$_X$, the contract limits the set of exceptions of both observers and the subject, 5–8 in Figure 1, of the event. Similarly, the subject `Savings` and the contract in Figure 12 illustrate a residual bug since the observers do not throw any unchecked exceptions, however the subject is still trying to catch them.

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 13 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 12 since it does not allow its observers to throw unchecked exceptions too. The subject `Savings` with the catch clause **`catch`** (**`RuntimeException`** e) {..} on line 13 and the contract in Figure 6 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*$_X$, 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 13 illustrate the occurrence of an exception stealer bug. According to the property Φ_{f1} of the bank account example, Section 2.2, 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 3–6 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 6 illustrate non occurrence of the exception stealer bug for the exception `RegDExc`. The contract for `WithdrawEv` in Figure 6 does not allow any try-catch expression to surround the

```
1 void WithdrawEv throws RegDExc{ ..
2  assumes{
3   try{
4    establishes next.acc().bal==old(next.acc().bal);
5    next.invoke();
6   } catch(Exception e){ }/*swallow*/
7  } .. }
```

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

invoke expression 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 12 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*$_{\mathcal{X}}$, (non) occurrence of all of them could be understood using *Ptolemy*$_{\mathcal{X}}$'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 translucid contract of `WithdrawEv`.

Bug Pattern	**Occurrence**	**Non Occurrence**
Throw without catch	✓	✓
Residual catch	✓	✓
Exception stealer	✓	✓
Path dependent throw	Not applicable to *Ptolemy*$_{\mathcal{X}}$	
Fragile catch	Not applicable to *Ptolemy*$_{\mathcal{X}}$	

Figure 14: *Ptolemy*$_{\mathcal{X}}$'s exception flow reasoning and understanding of AO bug patterns [6].

5.6 Application to Ptolemy

In this section we discuss the overhead of the application of *Ptolemy*$_{\mathcal{X}}$ 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 translucid 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*$_{\mathcal{X}}$. 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) [3] 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) [37] and Join Point Types (JPTs) [36] allow informal specification and reasoning about normal behaviors of aspects in AO. Pipa [40] 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 [25]. 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 [39] and the work of Malayeri and Aldrich [28] 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 [33], and the works of Jo *et al.* [18], Sinha *et al.* [35] 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 [38] and Shaner *et al.* [34] use greybox specifications for verification of mandatory calls. Rajan *et al.* [32] 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 $Ptolemy_X$ to address them. $Ptolemy_X$'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. $Ptolemy_X$'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 [30] and structural refinement for abnormal termination. For normal termination structural refinement implies refinement [2,34]. The second task would be application of $Ptolemy_X$ 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*.

8. REFERENCES

[1] M. Bagherzadeh, H. Rajan, and A. Darvish. On exceptions, events and observer chains. Technical Report 12-12, Iowa State U., 2012.

[2] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *AOSD'11*.

[3] E. Bodden, E. Tanter, and M. Inostroza. Safe and practical decoupling of aspects with join point interfaces. Technical Report TUD-CS-2012-0106, Technische U. Darmstadt.

[4] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, 1999.

[5] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: taming exceptional control flows in aspect-oriented programming. In *AOSD'08*.

[6] R. Coelho, A. Rashid, A. von Staa, J. Noble, U. Kulesza, and C. Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *PLoP'08*.

[7] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Asp. Comput.'98*, 10(3).

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

[9] S. Drossopoulou and T. Valkevych. Java exceptions throw no surprises. Technical report, Imperial College London, 2000.

[10] I. Figueroa and E. Tanter. A semantics for execution levels with exceptions. In *FOAL'11*.

[11] F. Filho, P. da S. Brito, and C. Rubira. Reasoning about exception flow at the architectural level. In *Rigorous Development of Complex Fault-Tolerant Systems'06*.

[12] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*.

[13] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. *TSE'05*, 31(4).

[14] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *FSE'98*.

[15] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91*.

[16] B. Jacobs, P. Muller, and F. Piessens. Sound reasoning about unchecked exceptions. In *SEFM'07*.

[17] B. Jacobs and F. Piessens. Failboxes: Provably safe exception handling. In *ECOOP 2000*.

[18] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe. An uncaught exception analysis for Java. *J. Syst. Softw.'04*, 72(1).

[19] C. B. Jones. Tentative steps toward a development method for interfering programs. *TOPLAS'83*, 5(4).

[20] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving aspect-oriented programs. In *FOAL'08*.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*.

[23] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05*.

[24] N. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI'09*.

[25] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *Softw. Eng. Notes'06*, 31(3).

[26] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica'95*, 32(8).

[27] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *TOPLAS 2000*, 22(2).

[28] D. Malayeri and J. Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques'06*.

[29] R. A. Maxion and R. T. Olszewski. Improving software robustness with dependability cases. In *FTCS'98*.

[30] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Com. Program.'87*, 9(3).

[31] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*.

[32] H. Rajan, J. Tao, S. M. Shaner, and G. T. Leavens. Tisa: A language design and modular verification technique for temporal policies in web services. In *ESOP'09*.

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

[34] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA '07*.

[35] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *ICSE'04*.

[36] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM'10*, 20(1).

[37] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM'10*, 20(2).

[38] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES'03*.

[39] 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*.

[40] J. Zhao and M. Rinard. Pipa: a behavioral interface specification language for AspectJ. In *FASE'03*.