Reasoning Tradeoffs In Implicit Invocation And Aspect Oriented Languages

José Sánchez

CS-TR-15-02

April 2015

Keywords: Modularity, program reasoning, tradeoffs, specification, verification, formal methods, Ptolemy language.

2012 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, logics of programs, pre- and postconditions, specification techniques;

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Computer Science
4000 Central Florida Blvd.
University of Central Florida
Orlando, Florida 32816, USA

REASONING TRADEOFFS IN IMPLICIT INVOCATION AND ASPECT ORIENTED LANGUAGES

by

JOSE SANCHEZ
B.S. Computer Science, University of Costa Rica, 1989
M.S. Computer Science, Costa Rica Institute of Technology, 1998

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2015

Major Professor: Gary T. Leavens

© 2015 José Sánchez

ABSTRACT

To *reason about* a program means to state or conclude, by logical means, some properties the program exhibits; like its correctness according to certain expected behavior. The continuous need for more ambitious, more complex, and more dependable software systems demands for better mechanisms to modularize them and reason about their correctness. The reasoning process is affected by the design decisions made by the developer of the program and by the features supported by the programming language used. Beyond Object Orientation, *Implicit Invocation* and *Aspect Oriented* languages pose very hard reasoning challenges. Important *tradeoffs* must be considered while reasoning about a program: modular vs. non-modular reasoning, case-by-case analysis vs. abstraction, explicitness vs. implicitness; are some of them. By deciding a series of tradeoffs one can configure a reasoning *scenario*. For example if one decides for modular reasoning and explicit invocation a well known object oriented reasoning scenario can be used.

This dissertation identifies various important tradeoffs faced when reasoning about implicit invocation and aspect oriented programs, characterize scenarios derived from making choices regarding these tradeoffs, and provides sound proof rules for verification of programs covered by all these scenarios. Guidance for program developers and language designers is also given, so that reasoning about these types of programs becomes more tractable.

To my beloved wife Dunia,

who by her devotion, strength and faith deserves all the credits;

and to my precious and brave children, Alejandra and José Pablo.

A mi amadísima esposa Dunia,

quien por su abnegación, fortaleza y fe merece todos los créditos,

y a mis preciosos y valientes hijos, Alejandra y José Pablo.

ACKNOWLEDGMENTS

This work was supported in part by Costa Rica's Universidad Nacional (UNA), Ministerio de Ciencia y Tecnología (MICIT) and Consejo Nacional para Investigaciones Científicas y Tecnológicas (CONICIT). Also by NSF grant CCF-1017334 titled "SHF: Small: Collaborative Research: Balancing Expressiveness and Modular Reasoning for Aspect-Oriented Programming." My gratitude to all these organizations and to the people at them that helped me through all this process.

My very special thanks to my advisor Dr. Gary T. Leavens, for his continuous support, sympathy and guidance. I also thank the other members of my committee: Dr. Damla Turgut, Dr. Sumit Kumar Jha and Dr. Heath M. Martin. My appreciation also goes to my lab-mates for their friendship and mutual encouragement.

I express my deep gratitude to my dear friends Marco and Blanca, and their lovely family, for embracing me, and my family, at the toughest times, and from then on.

My affection to my family (blood and in-law). Their caring and true love is always at the basis of our achievements. My deepest love to my mother, who could not wait for this moment, but is with me all the time.

Finally, all my love and gratitude to my wife Dunia and my children Alejandra and José Pablo. It's been a challenge for all of us, but our love and the powerful hand of God keep us even stronger.

TABLE OF CONTENTS

LIST OF FIGURES x

LIST OF TABLES xiii

CHAPTER 1: INTRODUCTION 1

1.1 Problem Definition 3

1.2 Scope 5

1.3 Motivation 6

CHAPTER 2: BACKGROUND 8

2.1 Reasoning and Modular Reasoning 8

2.2 Specifications and Specification Refinement 10

2.3 Proof Rules and their Soundness 13

2.4 Deductive Program Verification and Weakest Precondition Semantics 16

2.5 Behavioral Subtyping and Supertype Abstraction 17

2.6 Event Based Implicit Invocation Programming 18

2.7 Aspect Oriented Programming 19

CHAPTER 3: IMPROVING AND STATICALLY VERIFYING THE *PTOLEMY* IMPLICIT INVOCATION LANGUAGE 22

3.1 The Ptolemy Language 22

3.2 PtolemyRely: Separating Obligations of Subjects and Handlers 24

3.3 Static Verification of PtolemyRely Programs Using OpenJML 26

3.4 Methodology Oriented Approaches 28

CHAPTER 4: REASONING TRADEOFFS 30

4.1 Modular vs. Non-Modular Reasoning 30

4.2 Case analysis vs. Abstraction 34

4.3 Explicit Invocation vs Implicit Invocation 35

4.4 Explicit Announcement vs Implicit Announcement 36

CHAPTER 5: REASONING SCENARIOS AND THEIR PROOF RULES 38

5.1 Algebra of Specifications 38

5.2 The Reference Scenario 43

5.3 The Object Oriented Scenario 44

5.4 II/EA Scenarios 46

5.4.1 II/EA Full Delivery Scenario 46

5.4.2 II/EA Single Delivery Scenarios 48
5.4.2.1 II/EA PtolemyRely Modular Scenario 50
5.4.2.2 II/EA Ptolemy Modular Scenario 52
5.4.2.3 II/EA Behavior-Preserving Modular Scenario 55
5.4.2.4 II/EA Non-Modular Individual Refinement Scenario 57
5.5 AO Scenarios 62
CHAPTER 6: EVALUATION 65
6.1 Soundness of Scenarios' Proof Rules 65
6.1.1 Single Delivery II: PtolemyRely 65
6.1.2 Full Delivery II 79
6.1.3 Single Delivery II: Non-Modular Individual Refinement 88
6.2 Reasoning Cases 96
6.2.1 II/EA Full Delivery Reasoning 97
6.2.2 II/EA Ptolemy Reasoning 100
6.2.3 II/EA PtolemyRely Reasoning 105
6.2.4 Modular Behavior-Preserving Reasoning 111
6.2.5 Non-Modular Individual Refinement Reasoning 118

CHAPTER 7: CONCLUSION 124

7.1 Tradeoffs and Reasoning 124

7.2 Scenarios 126

7.3 Future Work 128

LIST OF REFERENCES 130

LIST OF FIGURES

1.1 Client-provider interaction 4

2.1 Program reasoning 8

2.2 Modular vs Non-Modular Reasoning 9

2.3 Hoare Logic assignment axiom and Sequence rule 13

2.4 Reasoning example 14

2.5 Applying proof rules 14

2.6 SOS of assignment ($ASSIGN_S$) and sequence (SEQ_S) 15

2.7 Behavioural Subtyping 18

2.8 Implicit Invocation example 19

2.9 AspectJ Loggin example 20

3.1 Ptolemy event example 23

3.2 PtolemyRely event example 25

3.3 Closure construct 28

4.1 Supertype Abstraction 34

5.1 Reference Scenario 43

5.2 OO Scenario 44

5.3 II/EA Full Delivery Mode 46

5.4 Full Delivery at Runtime 47

5.5 II/EA Single Delivery Mode 48

5.6 II/EA Single Delivery at Runtime 48

5.7 II/EA Single Delivery Abstraction 49

5.8 AO Scenario 62

5.9 AO Shadows 63

5.10 AO at Runtime 64

6.1 PtolemyRely's evaluation contexts and configuration. 66

6.2 PtolemyRely's Semantics. 67

6.3 Full Delivery Semantics. 80

6.4 Bill class 97

6.5 Full Delivery Example 98

6.6 Ptolemy Example 101

6.7 PtolemyRely Example 106

6.8 Modular Behavior-Preserving Example 111

6.9 Non-Modular Individual Refinement Example 118

LIST OF TABLES

2.1 Weakest precondition semantics 17

CHAPTER 1: INTRODUCTION

In software development, especially for large projects, there is always the need to modularize the system as a set of components that could be later integrated into bigger components and eventually into the final system. This modular approach always poses challenges for reasoning about the behavior of each component and the behavior of the integrated system. The reasoning process verifies that a piece of software satisfies a desired property, and could be done by the developer, by an automatic system or by a combination of both. The most important property to verify is the functional correctness of a piece of code with respect to its specification. This specification is usually expressed in a behavioral interface specification language (BISL) [28], like Eiffel [43], JML [37, 38, 13] or Spec# [9], using constructs like pre- and post-conditions, invariants and assertions. The reasoning process itself can be modular or non-modular. In the first case, the modules are reasoned about one at a time, using the specification of other modules when required. In the non-modular case all the modules are reasoned about together, re-reasoning about the modules every time they are referred to. Although modular reasoning is desirable, sometimes it implies losing some information, so the tradeoff "modular vs. non-modular" should be considered.

Traditionally, in Object Oriented (*OO*) programming, integration among components is performed by making the client component explicitly call methods from the provider component. The place in the client code where the method is called is explicit (Explicit Announcement, *EA*) and the provider's method to be called is also explicitly known (Explicit Invocation, *EI*). In this case reasoning about the client code can be performed using the specification, instead of the implementation, of the called methods. A more complex situation arises with dynamic dispatching, where the actual implementation that is invoked at a method call is determined dynamically, at runtime. For example, in a hierarchy of classes various sub-classes might have their particular implementation of a method declared in an ancestor class. A method call using a reference declared with the type

of the ancestor class can execute any of the various implementations. That situation shows a reasoning tradeoff: to do a case analysis considering the behavior of each implementation or to use the more abstract behavior in the ancestor class (for modular reasoning).

More sophisticated and flexible integration approaches are provided by Implicit Invocation (*II*) [27, 47, 62, 49], in Event Based programming, and Implicit Announcement (*IA*) and Implicit Invocation, in Aspect Oriented (*AO*) [35, 23] programming.

In II languages, the client component announces or broadcasts events, instead of calling specific methods. The system *implicitly* invokes the target components that have been registered for those events. This loosely coupling between components eases system evolution and reuse. Target components can be added, changed or removed without modifying the client. Reasoning about the client is more involved, since it has no references to the target components that will be invoked.

AO languages address the challenge of modularizing functionality that is scattered through many components, and tangled with their main logic. In AO programming, this functionality is encapsulated into aspect modules, and implicitly woven (restored) at the required points. An expressive mechanism may select (quantify) these points. The client component is not only oblivious of *what* aspects might be applied but also of the places *where* they are invoked. That makes the reasoning task even harder, showing that “explicitness vs. implicitness” is an important tradeoff.

In reasoning OO, II and AO programs different tradeoffs are faced: modular vs. non-modular reasoning, case analysis vs. abstraction, explicitness vs. implicitness, etc. In this work that type of tradeoffs are considered while formalizing proof rules for reasoning about programs written in these languages. These formalizations are useful not only for program verification but also as guidance for language designers and developers, when choosing the features they use and how to reason about them.

1.1 Problem Definition

The problem I aim to solve in this work is to show how to do formal reasoning for specific scenarios in implicit invocation (II) and aspect oriented (AO) languages, and to provide guidance to language designers and developers about programming methodology for this languages. The scenarios correspond to different configurations of important tradeoffs faced when reasoning about II and OO programs.

The main parts of this problem are:

- To identify and justify different tradeoffs that are faced when reasoning about a client component that is advised by II or AO provider components.
- To formally characterize scenarios derived from making choices regarding these tradeoffs.
- To provide sound proof rules for verification in the different scenarios.

Important tradeoffs include modular vs. non-modular reasoning, case analysis vs. abstraction, explicitness vs. implicitness. For doing modular reasoning each module should have a specification. The module is reasoned about once and for all against that specification. When reasoning about modules that reference others modules, the referenced modules' specifications are used, instead of the modules themselves. In some situations, like dynamic dispatching and implicit invocation, different behaviors may apply at a certain point. Doing modular reasoning would require to use a specification that abstracts the different behaviors, maybe losing important information and weakening the reasoning conclusions. In this cases modularity could become a tradeoff, in favor of stronger conclusions in a case-by-case analysis.

In the traditional Object-Oriented (EI/EA) approach the client component, at explicit places (EA), calls explicit methods (EI) of the provider component. A modular verification approach can be

used. The called method is first verified against the specification given for that method, as found in the receiver's static type, and this specification is used later in the verification of the client component. The basic idea of this OO case is illustrated in Figure 1.1(a). Polymorphic method calls can be handled modularly using supertype abstraction [41], enabled by behavioral subtyping [2, 42, 39]. Otherwise a case-by-case analysis would be required.

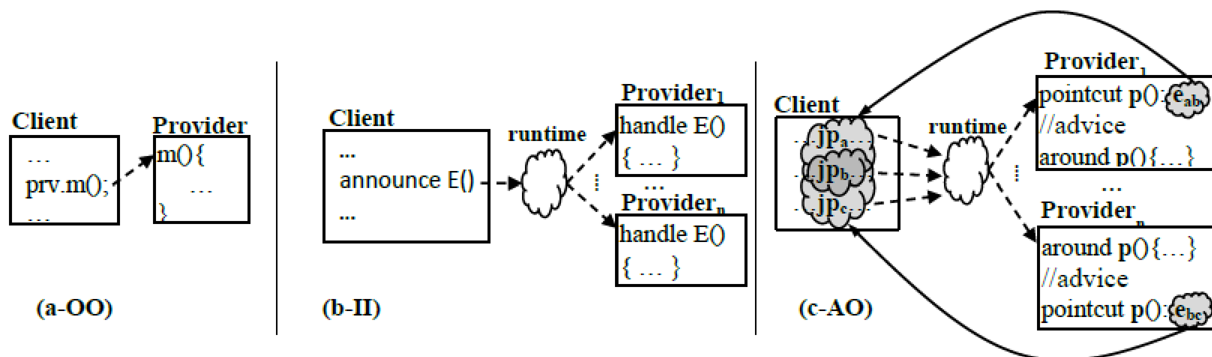

Figure 1.1: Client-provider interaction: (a) Object Oriented, (b) Implicit Invocation, (c) Aspect Oriented

In event based II/EA, the client (subject) component announces or broadcasts events at explicit places (EA), using some type of **`announce`** construct. The system implicitly invokes (II) the registered methods (handlers) in the provider components. There are two reasoning challenges. First, the client code does not know what handler, or handlers, will be executed in response to the event announcement. There is no concrete specification to reason about the event announcement and so the client depends on the specification of all possible handlers. Second, an **`invoke`** statement in the body of a handler will execute the next handler or the announced code. Again there is no concrete specification to reason about the **`invoke`** statement and so the handlers depend on each other and on the announced code. Figure 1.1(b) shows the II approach.

Several issues arise. Should announced-code specifications determine handler's specifications or should it be the other way around? How does that determine the reasoning of the client code and the handlers? How to preserve the original client reasoning when event announcements are

added? What criteria should a developer or a tool use to determine or infer the specifications for the handlers? What are the pros and cons of enforcing modular reasoning? These issues should be analysed and resolved to take advantage II/EA while being able to reason about the correctness of the system.

Aspect oriented programming takes implicitness (obliviousness) to the extreme using II/IA. The client component makes no explicit mention (IA) of any unforeseen functionality (e.g. cross-cutting concerns) that may be added by any provider component, nor marks any specific place in its code that announces any possibility of extension. Instead, pointcuts inside aspects (provider components) select some join points (shadows) in the client code, where advice from the aspect is added. Considering the notions of modular reasoning (sec. 2.1) and obliviousness (sec. 2.6) adopted in this work, modular reasoning is not in general possible under complete obliviousness. Even if the join-points picked by a pointcut are identified, the situation falls back to the previous case of II/EA, exhibiting the same issues mentioned before. The AO case is also shown in Figure 1.1(c).

In this thesis a formal design workbench for verification of II and AO programs is defined. The design variables, design tradeoffs and possible scenarios are identified and formally characterized, and the achievable verification results for each case are established my means of sound proof rules.

1.2 Scope

This work aims at showing how to do formal reasoning for specific scenarios in implicit invocation and aspect oriented languages based on Java language. The general scenario is to reason about a client component, that has a set of advisable blocks, and a set of provider components, that embody the advice or handling methods. The following limits in scope apply.

- The client and provider components will be assumed to execute in a sequential fashion. Although interesting and important, to simplify the formal analysis concurrent execution will not be considered.
- Only *around* advice will be considered, since for reasoning purposes *before* and *after* advice can be handled as special cases of it.
- Control flow pointcuts, like *clfow*, will not be considered because their intrinsic non-modular nature complicates their reasoning.

1.3 Motivation

The continuous need for more ambitious, more complex, and more dependable software systems demands for better mechanisms to modularize them and verify their correctness. Modularity not only makes complexity tractable but also improves maintainability and reusability. On the other hand, formal methods, for example satisfiability-based program reasoning, help in improving the dependability of the systems, by formally verifying their correctness.

Implicit Invocation and Aspect Oriented programming languages provide attractive features for better modularizing systems. At the same time, these features introduce difficulties for reasoning about the behavior of the system. This section further emphasizes the value of these features from a software engineering perspective and the need for guidance and discipline in their use.

II allows for loose coupling between the client and provider components. That improves flexibility and maintainability, as new provider components can be added just by registering them, without any change in the client component. Despite its advantages, II is not ideal for dealing with crosscutting concerns. A concern, like logging or authorization, that cut across a system could be implemented using II: at any place in the client code where the crosscutting concern is required an

event announcement can be placed. The provider component that implements the concern will be implicitly invoked at all these points. The problem is that the event announcement will be scattered throughout the system and tangled with the main logic of all the modules. Aspect Orientation takes a step further for implementing crosscuting concerns by means of its II/IA mechanism. Inside an aspect module, an advice implements the crosscutting concern and a pointcut selects (quantifies) the join points in the client component where the concern needs to be applied. An important issue of this approach is that the provider component (aspect), specifically its pointcuts, have explicit references to the client or base component. That not only increases the coupling between the components but also causes the *fragile poincut* problem [54], where even small changes in the client component may prevent the provider component from being invoked.

II and AO both provide mechanisms to modularize software systems, the first necessity mentioned before for implementing complex and dependable systems. Nevertheless these mechanisms introduce difficulties for reasoning about the behavior of the system to ensure its dependability, the second necessity mentioned before. As will be shown in chapter 3, many proposals [58, 11, 31, 54] have been made in the last years to correct some of the problems of AO and improve the ability to reason about programs. Most of these proposal provide some sort of interface type that further decouples client (base code) and provider (aspect) components. Some of them [54] also include language features to specify and verify the obligations of both the clients and providers.

There are many trade-offs, complexities and mutual dependencies when reasoning about client and provider components in implicit invocation and aspect oriented systems. Even if a chosen language has some specification and verification features, it is necessary to carefully explore the design space to find the best ways to use them for reasoning about program written in these languages. The clear benefits of modularizing mechanisms like II and AO and the lack of strategies and discipline that guide language designers and developers on their use constitute strong motivations for addressing these issues, as I am doing in this work.

CHAPTER 2: BACKGROUND

Some concepts are required to more precisely define the research problem and the proposed approach. In this section a few of them are introduced, like program reasoning, specifications and refinement, deductive verification and weakest precondition semantics, behavioral subtyping and supertype abstraction, implicit invocation in event-based programming, and implicit announcement and invocation in Aspect Oriented programming.

2.1 Reasoning and Modular Reasoning

To *reason about* a program means to state or conclude, by use of deduction, some properties the program exhibits; like correctness, according to certain expected behavior, or a desired level of performance. These properties can be asserted by empirical observation or by a more formal logical analysis. For example, by reasoning about the code in Figure 2.1 it can be concluded that `M2` computes the minimum of `x` and `y`.

```
1 int M2(int x, int y){
2   if (x<y) return x;
3     // if x<y then M2(x,y)==x
4   else return y;
5     // if y<=x then M2(x,y)==y
6 }
7 // reasoning conclusion: M2(x,y)==min(x,y)
```

Figure 2.1: Program reasoning: `M2` computes the minimum of `x` and `y`

Usually programs are constructed in a modular way: the program is divided into sub-programs (modules) that are later combined (composed) to get the desired functionality. *Modular reasoning* is the process of reasoning about a program one module at a time [48], using the established

properties of already reasoned about modules when required, instead of re-reasoning about them. On the contrary, non-modular or whole program reasoning needs to consider the whole program (all the used modules) for establishing the desired properties. The programs in Figure 2.2 illustrate the modular and non-modular reasoning.

```
1  // property PM2:M2(x,y)==min(x,y)
2  /*@ ensures
3  (\result==x || \result==y)
4  && \result<=x && \result<=y;
5  @*/
6  int M2(int x, int y){
7   if (x<y) return x;
8   else return y;
9  }
10
11 // property PM3:M3(a,b,c)==min(a,b,c)
12 /*@ ensures
13 (\result==a || \result==b || \result==c)
14 && \result<=a && \result<=b && \result<=c;
15 @*/
16 int M3(int a, int b, int c){
17  int d,e;
18  d=M2(a,b); // PM2 => d==min(a,b)
19  e=M2(c,d); // PM2 => e==min(c,d)
20    //=> e==min(c,min(a,b))
21  return e;  // => e=min(a,b,c)
22 }
```

(a) modular reasoning

```
1  int M2(int x, int y){
2   if (x<y) return x;
3   else return y;
4  }
5
6  int M3(int a,int b,int c){
7   int d,e;
8
9   d=M2(a,b);
10   // if (a<b) d=a
11   // else d=b;
12   // => d==min(a,b)
13
14  e=M2(c,d);
15   // if (c<d) e=c
16   // else e=d;
17   // => e==min(c,d)
18
19   // => e==min(c,min(a,b))
20
21  return e;//=>e==min(a,b,c)
22 }
23 // reasoning conclusion:
24 // M3(a,b,c)== min(a,b,c)
```

(b) non-modular reasoning

Figure 2.2: Modular vs Non-Modular Reasoning

In the modular case (a) the module (function) `M2` is reasoned about and the property `PM2` is established, asserting that `M2` computes the minimum of its two parameters. This property is used at each call to `M2` (lines 18,19) while reasoning about module `M3`. Noticeably only the property `PM2` is required, not the body of module `M2`. In the non-modular case (b), at each call to `M2` its body is re-reasoned about (lines 9-12 and 14-17). That requires more work and needs the body of `M2` to be

available. Such reasoning would be difficult for a call to an interface method in Java, for example.

Along existing ideas [48, 36, 15], in this work *modular reasoning* is understood as being able to establish properties of a module just by considering its interface, specification and implementation; and the interfaces and specifications of modules referenced by them. A module B is *referenced* by another module A if B is explicitly named or lexically contains A. If it were necessary to consider other modules not referenced or to consider the implementation of other modules, instead of just their interfaces and specifications, then the reasoning is non-modular (or "whole-program" reasoning).

2.2 Specifications and Specification Refinement

A functional specification declares the expected behavior of a *program* (block of statements, like a method or a fragment of it). The JML specification **`requires`** P **`ensures`** Q indicates that the expected behavior of a program is that whenever it starts execution in a state satisfying *precondition* P, and terminates, then the final state satisfies *postcondition* Q. This is the partial correctness expressed by the Hoare [30] triple $\{P\}S\{Q\}$, where S is the program's code. For example, the program $S = x := y + 1$ satisfies the specification **`requires`** $y > 0$ **`ensures`** $x > 1$, written in Hoare logic as $\{y > 0\}S\{x > 1\}$, but the program $S = x := y$ does not satisfy that specification. Specifications can be used as contracts against which to verify implementations, the actual programs. They can also be used to reason about the execution of programs. For example, if a program S, that is guaranteed to satisfy the previous specification, is executed from a state where $y > 0$ then one can conclude that at termination $x > 1$.

Since functional specifications do not establish the locations a program can modify, certain expected deductions about the program execution can not be made. For example after executing

a program S, like $S = x := y + 1$, that satisfies the triple $\{y > 0\}S\{x > 1\}$ one cannot conclude that the condition $y > 0$ is preserved. That is due to the fact that maybe S, although satisfying the specification, may also alter y, invalidating the new conclusion. If, on the contrary, the specification only allows x to be modified, then the conclusion $y > 0$ would be valid. As shown by the previous example, knowing the locations a program is allowed to modify, also called *framing* a program [12], is important for reasoning about it. The framed specification format **requires** P **modifies** ε **ensures** Q indicates that a conforming program, S, is partially correct, as above, and only modifies locations in ε, $mods(S) \subseteq \varepsilon$. To define this precisely, the function $mod(S)$ returns the list of locations potentially modified by S. A program *modifies* a location if (a) the location exists in the program's pre-state and (b) the location is assigned to during the program's execution. The Hoare-like formula $\{P\}S\{Q\}[\varepsilon]$ expresses both partial correctness and framing, where ε is the *frame* of S. For instance, the previous functional specification can be extended by a framing specification yielding the specification **requires** $y > 0$ **modifies** x **ensures** $x > 1$, that corresponds to the formula $\{y > 0\}S\{x > 1\}[x]$. This specification allows one to conclude that, after execution of S, not only $x > 1$ holds but also $y > 0$ does.

A specification can be expressed as an ordered triplet (P, Q, ε), where P is the precondition predicate, Q the postcondition predicate and ε is the frame. The precondition is evaluated on the state of the system when the program starts execution, called the pre-state, and the postcondition on the state at program termination, the post-state. However, as the postcondition may also refer to values in the pre-state, for example to express that the value of one variable in the post-state must be greater than its value in the pre-state, then the post-condition will depend on both states. This situation where the predicates can range over two states is referred as two-state specifications. In a postcondition the keyword **old** will be used to refer to the value of an expression in the pre-state. For example the triplet $(l \leq r,\ \mathbf{old}(l) \leq m \wedge m \leq \mathbf{old}(r),\ \{m\})$ specifies a program S that, starting from a state where $l \leq r$, computes in m a value that lies between the original values of l

and r and only modifies m.

The *refinement* relation, $\sqsupseteq$, is used to describe both the satisfaction relation between a program and a specifications and the refinement relation between a specification and a *more-defined* specification, as detailed in definition 1.

Definition 1. *(Refinement)*

i. specification refinement (satisfaction) by a program:

$S \sqsupseteq (P, Q, \varepsilon) \Leftrightarrow \{P\}S\{Q\}[\varepsilon]$

ii. specification refinement by another specification:

$(P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon) \Leftrightarrow [\forall\, program\ S,\ (S \sqsupseteq (P', Q', \varepsilon')) \Rightarrow (S \sqsupseteq (P, Q, \varepsilon))]$

For example, the following program refines the given specification:

[1] $\{m := (l + r)/2\} \sqsupseteq (l \leq r,\ \mathbf{old}(l) \leq m\ \wedge\ m \leq \mathbf{old}(r),\ \{m\}),$

and similarly this is an example of a specification refining another specification:

$$(l + 1 \leq r - 1, \mathbf{old}(l) < m \wedge m < \mathbf{old}(r), \{m\}) \sqsupseteq$$
$$(l + 1 < r - 1, \mathbf{old}(l) \leq m \wedge m \leq \mathbf{old}(r), \{m\}).$$

Specifications can be refined in various ways [44]:

- weakening the precondition: $(P \Rightarrow P') \Rightarrow ((P', Q, \varepsilon) \sqsupseteq (P, Q, \varepsilon))$,
- strengthening the postcondition: $(Q' \Rightarrow Q) \Rightarrow ((P, Q', \varepsilon) \sqsupseteq (P, Q, \varepsilon))$,
- restricting change: $(\varepsilon' \subseteq \varepsilon) \Rightarrow ((P, Q, \varepsilon') \sqsupseteq (P, Q, \varepsilon))$,
- introducing fresh local variables: $fresh(x, (P, Q, \varepsilon)) \Rightarrow ((P, Q, \varepsilon \cup \{x\}) \sqsupseteq (P, Q, \varepsilon))$, or
- by a combination of the above [14].

For example, if we take the program $S = \{\mathbf{if}\ (x > 0)\{x := 1; y := 2\}\ \mathbf{else}\ \{x := 1\}\}$, then

[1] In the expression $(l + r)/2$, $+$ and $/$ represent integer arithmetic.

$S \sqsupseteq (x \leq 0,\ x = 1,\ \{x\})$ but $S \not\sqsupseteq (true,\ x = 1,\ \{x\})$, as the framing is not guaranteed by the latter's precondition.

2.3 Proof Rules and their Soundness

A *proof rule* is an axiom or rule of inference [30] that can be used for proving properties of computer programs, like its functional correctness against a given specification. A set of proof rules form a *logic* for the corresponding language. A logic is *sound* if every judgment that is probable using its proof rules is actually valid; where this validity is defined with respect to a formal semantics of the given programming language. The soundness of a logic is fundamental. It ensures that whenever one reasons about a program using that logic the program actually exhibits the proven behavior.

Proof rules are usually written using the format: $\dfrac{A_1, \ldots A_n}{C}$, where the A_i's are the premises and C is the conclusion. For example, in Hoare logic the assignment axiom ($ASSIGN_R$) and the sequential composition inference rule (SEQ_R) are as follows:

$$\text{(ASSIGN}_R\text{)}\ \frac{}{\{P[e/x]\}x := e\{P\}} \qquad \text{(SEQ}_R\text{)}\ \frac{\{P\}S_1\{R\}, \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}$$

Figure 2.3: Hoare Logic assignment axiom and Sequence rule

The assignment axiom states that for proving that a condition, P, holds after an assignment, $x := e$, it is required to prove that the same condition, with the variable substituted by the expression, $P[e/x]$, hods before the assignment. The sequence inference rule allows one to prove that a sequence, $(S_1; S_2)$, satisfies a specification (P, Q) by proving that S_1 satisfies a specification, (P, R), whose postcondition matches the precondition of a specification, (R, Q), satisfied by S_2. For exam-

ple, using those two rules it can be proved that the program in Figure 2.4 satisfies its specification.

```
1 //@ requires x≥1;
2 //@ modifies x;
3 //@ ensures x≥3);
4 S() {
5   x=x+x;
6   x=x+1;
7  }
```

Figure 2.4: Reasoning example

As shown in Figure 2.5, the SEQ_R rule is used to split the proof into two sub-proofs and then the $ASSIGN_R$ axiom is used to prove each one of them.

$$\text{SEQ}\ \dfrac{\text{ASSIG}\ \dfrac{\{(x+x)\geq 2\}x := x+x\{x\geq 2\}}{\{x\geq 1\}x := x+x\{x\geq 2\}} \qquad \text{ASSIG}\ \dfrac{\{(x+1)\geq 3\}x := x+1\{x\geq 3\}}{\{x\geq 2\}x := x+1\{x\geq 3\}}}{\{x\geq 1\}x := x+x;\ x := x+1\{x\geq 3\}}$$

Figure 2.5: Applying proof rules

As already said, the soundness of the proof rules must be proven against the formal semantics of the programming language. Usually this semantics is expressed using the well known *structural operational semantics (SOS)* [52]. A mapping, Σ, models the *state* of memory, mapping each location to it's corresponding value. A *configuration*, $\langle \mathbb{E}[S], \Sigma \rangle$, describes the execution state, containing at least the program to execute, S, and the current memory state, Σ. The *semantics* for sentences is expressed as a *evaluation* relation between configurations $\langle \mathbb{E}[S], \Sigma \rangle \hookrightarrow \langle \mathbb{E}[S'], \Sigma' \rangle$, meaning that, in state Σ, executing one step of statement S leads to state Σ' and the remaining statement to execute is S'. The SOS of a language takes the form of a set of inference rules which define the valid transitions of a program in terms of the transitions of its components. For example, Figure 2.6 shows some inference rules describing the semantics of assignment and sequencing in a typical imperative programming language.

$$\text{(ASSIGN}_S\text{)}\quad \frac{[\![e]\!]_\Sigma = v}{\langle \mathbb{E}[x := e], \Sigma\rangle \hookrightarrow \langle \mathbb{E}[\texttt{skip}], \Sigma[x \mapsto v]\rangle} \qquad \text{(SEQ}_S\text{)}\quad \frac{\langle \mathbb{E}[S_1], \Sigma\rangle \hookrightarrow \langle \mathbb{E}[S_1'], \Sigma'\rangle}{\langle \mathbb{E}[S_1; S_2], \Sigma\rangle \hookrightarrow \langle \mathbb{E}[S_1'; S_2], \Sigma'\rangle}$$

Figure 2.6: SOS of assignment ($ASSIGN_S$) and sequence (SEQ_S)

In demonstrating the soundness of proof rules, it is often required to demonstrate the validity of a Hoare triple of the form $\{P\}S\{Q\}$ with regard to the semantics of the language. The key equivalence is:

$$\{P\}S\{Q\} \Leftrightarrow \forall \Sigma, \Sigma' \bullet ([\![P]\!]_\Sigma \wedge \langle \mathbb{E}[S], \Sigma\rangle \hookrightarrow^* \langle \mathbb{E}[skip], \Sigma'\rangle) \Rightarrow [\![Q]\!]_{\Sigma'} \tag{2.1}$$

For example, for demonstrating the validity of the $ASSIGN_R$ proof axiom one can use the $ASSIGN_S$ semantic rule as follows.

Proof: [$ASSIGN_R$ proof rule]

$\langle$by hypothesis$\rangle$

$[\![P[e/x]]\!]_\Sigma \wedge \langle \mathbb{E}[x := e], \Sigma\rangle \hookrightarrow \langle \mathbb{E}[skip], \Sigma'\rangle$

$\Longrightarrow$ $\langle$by $ASSIGN_R$ semantic rule$\rangle$

$([\![P[e/x]]\!]_\Sigma) \wedge ([\![e]\!]_\Sigma = v) \wedge (\Sigma' = \Sigma[x \mapsto v]) \wedge$

$\Longrightarrow$ $\langle$by value substitution$\rangle$

$([\![P[v/x]]\!]_\Sigma) \wedge (\Sigma' = \Sigma[x \mapsto v])$

$\Longrightarrow$ $\langle$by logic$\rangle$

$([\![P]\!]_{\Sigma'})$

$\Longrightarrow$ $\langle$by equivalence 2.1$\rangle$

$\{P[e/x]\}x := e\{P\}$ ∎

The previous example illustrates how to demonstrate the soundness of proof rules with regard to

the semantics of the language.

2.4 Deductive Program Verification and Weakest Precondition Semantics

Verifying a program means certifying that it conforms to its specification in all possible executions. *Deductive verification* is one approach for doing program verification. Out of a program and its specifications, a collection of mathematical assertions, called *verification conditions* (VC's), are derived. The truth of these assertions must guarantee that the program satisfies its specifications. A theorem prover[2] is typically used to decide the truth of the VC's and so to verify whether the program satisfies its specifications. In this dissertation I will apply deductive verification to reason about programs based on AO-like interfaces.

For a program, S, there are, in general, many specification (P, Q, ε) that this program satisfies, $S \sqsupseteq (P, Q, \varepsilon)$. For example the program $\{x := x + 1;\}$ satisfies the specifications $(x \geq 1,\ x \geq 2,\ \{x\})$, $(true,\ x > old(x),\ \{x\})$, $(x > 1,\ x \geq 2,\ \{x\})$, etc. If one fixes the postcondition Q there are still many preconditions P such that the program S satisfies the specification (P, Q, ε). For postcondition $Q \equiv (x \geq 2)$, both preconditions $P_1 \equiv (x \geq 1)$ and $P_2 \equiv (x > 1)$ form a specification $(P, Q, \{x\})$ satisfied by S, however P_1 is weaker than P_2, as $P_2 \Rightarrow P_1$.

Given program S, postcondition Q and frame ε, the *weakest liberal precondition* of S with respect to Q and ε, $wlp(S)(Q, \varepsilon)$, is the weakest precondition P such that $\{P\}S\{Q\}[\varepsilon]$ is valid (noticeably $\{wlp(S)(Q, \varepsilon)\}S\{Q\}[\varepsilon]$ is valid). The semantics of a program S can then be expressed as a *weakest precondition predicate transformer*[3], where the "meaning" of the program S is expressed as a function $wlp(S)(\cdot)$ that transforms a predicate, the postcondition, into another predicate, the precondition. Table 2.1 shows the semantics of some commands using weakest precondition.

[2] A theorem prover is a computer program that determines if a set of assertions are provably true.

[3] Strongest postcondition is another way of predicate transformer semantics

Table 2.1: Weakest precondition semantics

Command	S	$wlp(S)(Q, \varepsilon)$
Assignment	$x := e$	$Q[x \mapsto e] \wedge x \in \varepsilon$
Sequence	$S_1; S_2$	$wlp(S_1)(wlp(S_2)(Q, \varepsilon), \varepsilon)$
Call	$p(e)$	$pre[x \mapsto e] \wedge \forall y(post[x \mapsto e][\varepsilon' \mapsto y] \Rightarrow Q[\varepsilon' \mapsto y]) \wedge \varepsilon' \subseteq \varepsilon$, if $(pre, post, \varepsilon')$ is the specification for method $p(x : T)\{S\}$

Weakest precondition semantics can be used to construct the verification conditions of a program, for doing deductive program verification. To verify $\{P\}S\{Q\}[\varepsilon]$ one computes $wlp(S)(Q, \varepsilon)$ and construct the verification condition $VC = (P \Rightarrow wlp(S)(Q, \varepsilon))$, whose proof can be delegated to a theorem prover. If VC holds then $\{wlp(S)(Q, \varepsilon)\}S\{Q\}[\varepsilon]$ is valid and as a consequence $\{P\}S\{Q\}[\varepsilon]$ is also valid, as desired.

2.5 Behavioral Subtyping and Supertype Abstraction

In Object Oriented programming, types, for example classes, can be organized in a hierarchical subtype relationship. From a behavioral point of view it is desirable that the subtype relationship also conveys a certain relationship among type specifications [42]. Instances of the subtype can be used in places where instances of the supertype are expected, generating *subtype polymorphism*. For example, if a method p has a parameter x of type A (p has the form $p(x : A)\{...\}$) and B is a subtype of A then the method can be invoked passing an instance of type B, like in $p(newB())$. This situation is illustrated in Figure 2.7

The type associated with an identifier at compile time is called its *static* type, while the (most specific) type of the actual instance referenced by the identifier at run time is called its *dynamic* type. In the example in Figure 2.7, the static type of parameter x is A, but when the method p is called, $p(newB())$, x's dynamic type is B. If the body of method p invokes method $x.m()$, as in $p(x : A)\{...;\ x.m();\ ...\}$, then there are at least two methods that could be executed: $A.m()$ or

$B.m()$. *Dynamic dispatch* selects that method based on the dynamic type of the callee. This poses a reasoning challenge, should the method call $x.m()$ be verified using the method's specification for the supertype, $A.m()$, or for the subtype, $B.m()$.

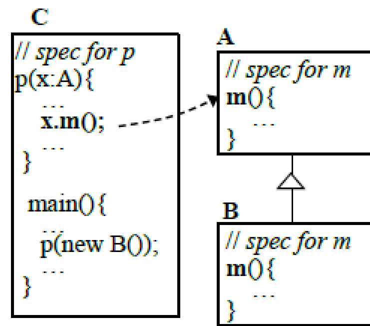

Figure 2.7: Behavioural Subtyping

The *behavioral Subtyping* discipline [2, 42, 39] solves this problem by requiring that the specification for a method in a subtype refines the specification of the corresponding method in the supertype that it overrides. In the example, $(P^B_m, Q^B_m, \varepsilon^B_m) \sqsupseteq (P^A_m, Q^A_m, \varepsilon^A_m)$, where $(P^A_m, Q^A_m, \varepsilon^A_m)$ is the specification of $A.m()$ and $(P^B_m, Q^B_m, \varepsilon^B_m)$ is the specification of $B.m()$. Behavioral Subtyping allows one to reason using *Supertype Abstraction* [41]. With Supertype Abstraction method calls are reasoned about using the specification in the static type of the receiver expression, $(P^A_m, Q^A_m, \varepsilon^A_m)$ in the example. In this way the program is kept valid no matter what are the dynamic types involved, as long as they are behavioral subtypes of the supertype in question.

2.6 Event Based Implicit Invocation Programming

In *Event Based Implicit Invocation (II)* [27, 47], the client (subject) component announces or broadcasts events at explicit places, using some type of **`announce`** construct (or feature). The system implicitly invokes the registered methods (handlers) in provider components. Implicit invocation improves flexibility and maintainability, for example new provider components can be added with-

out changing the client component. For a programming language to support implicit invocation it must provide features to declare events, including the context information they carry, to announce events and to bind (register) handlers to events. It also needs run-time support that implements the actual delivery of events, by implicitly invoking the registered handler methods. Figure 2.8 illustrates these concepts using a hypothetical Java-like Implicit Invocation language.

```
1  public class ClientComponent {
2   event LogginEvt(Customer c);
3
4   void process(Customer c){
5    ...
6    c.notify();
7    announce LogginEvt(c);
8    ...
9   }
10 }
11 public class ProviderComponent {
12  void handler(Customer c){
13   ...
14   Logger.log(c);
15  }
16  when LogginEvt do handler;
17
18  ProviderComponent(){
19   register(this);
20  }
21 }
```

Figure 2.8: Implicit Invocation example

2.7 Aspect Oriented Programming

Aspect Oriented (AO) programming adds new features to modularize cross-cutting concerns. A cross-cutting concern, like logging or authorization, is a conceptual requirement whose implementation could be scattered throughout the system and "tangled" with the main logic of its modules

(*base code*). AO aims to encapsulate these concerns into “aspects” and weave them at the required points. *Join points* correspond to predefined events in the execution of a program, like method calls or field accesses, and *join point shadows* are the actual points in the modules’ code where these events occur. *Pointcuts* are sets of join points. Aspects in *AspectJ* language include method-like blocks of code, called *advice*, that implement the cross-cutting concerns, and *pointcut descriptors*, that declaratively select a set of join points where to apply the advice (*quantification*).

```
1  public aspect Loggin {
2    ...
3    pointcut updating(): call(* *.update*(..)); //pcd
4    before(): updating()  {
5      logger.log("updating method called: "+time());
6  } }
7  public aspect Notification{
8    ...
9    pointcut notifying(): call(* *.notify()); //pcd
10   void around(): notifying()  {
11     if (notificationEnabled()){
12          proceed();
13     }
14 } }
15 public class base {
16   ...
17   void process(Customer c, Account a) {
18      c.updateAccount(a); // matches "updating" pcd
19      int balance=(int)(0.1*a.balance());
20      a.updateBalance(balance); // matches "updating" pcd
21      a.print();
22      c.notify(); // matches "notifying" pcd
23 } }
```

Figure 2.9: AspectJ Loggin example

Advice could be executed **`before`**, **`after`** or **`around`** the join point. **`Around`** advice is executed instead of the join point. In the body of the advice, a **`proceed`** instruction executes other advice if applicable or the original join point otherwise.

The example in Figure 2.9 illustrates all these concepts in the *AspectJ* language. Aspect `Loggin`

(lines 1-6) has the *updating* pointcut (line 3) and a piece of advise (lines 4-6), that will be executed at any join point matching the pattern in the pointcut. The pattern (line 3) matches calls to methods with any return type (first `*`), that belong to any class (`*.`), whose name starts with the word *update* followed by any sequence (`update*`) and has any parameters (`...`). This pattern matches the method calls at lines 18 and 20 in the base code. Then, before the actual call to these methods, the piece of login advice (line 5) will be executed. Aspect `Notification` (lines 7-14) is similar, but it has an **`around`** piece of advice. As its pointcut matches the `notify()` method call in the base code (line 22) then the advise body (lines 11-13) will be executed *instead of* that method call. The **`proceed`** instruction in the advice (line 12) will go on with the actual execution of the original `notify()` method call. As that **`proceed`** instruction is inside a conditional statement, only if the condition (line 11) holds the original method call will be executed.

As proposed by Filman and Friedman [25] *obliviousness* is a defining characteristic of AO programming. It "states that you can't tell that the aspect code will execute by examining the body of the base code" [24]. In the text of the client code there is no reference to any advice that could be applied. If one inspects the base code (lines 15-23) there is no explicit indication of where or what advice could be applied. That makes reasoning about AO programs a challenge.

CHAPTER 3: IMPROVING AND STATICALLY VERIFYING THE *PTOLEMY* IMPLICIT INVOCATION LANGUAGE

Different approaches, like JPT [58], JPI [11], EJP [31] and Ptolemy Event Types [54] propose language support for certain types of interfaces that mediate the interactions between the client and provider components, and so enable modular type checking and/or modular reasoning about AO-like languages. Other approaches like Co-AOP [32, 33] and XPIs [59, 60] pay more attention to methodology and less to language features.

3.1 The Ptolemy Language

The Ptolemy approach explicitly addresses the problem of specification and verification, and has been the starting point of some previous work [55, 56] by this author.

The language Ptolemy [54] is an extension of Java with support for the implicit invocation architectural style [27, 47], plus some aspect oriented features. It also incorporates translucid contracts [7, 5] that provide functional specification features [1] and control effects reasoning.

In support for II, Ptolemy provides features to declare *event* types (Fig. 3.1-a, line 1), to explicitly **`announce`** event occurrences (line 7) and to **`register`** (line 21) and bind (line 18) handlers with events. Bindings associate a handler method to the set of events named in a **`when-do`** clause. Ptolemy also provides the run-time support that implements the delivery of events, by implicitly invoking the registered handler methods. Ptolemy event types are declared independently from the client components, which announce them, and from the provider components that handle them. An event type can be announced by many clients and can be handled by many providers. Event types

[1] Specifications in Ptolemy do not include framing, so they are of the form (P, Q) instead of (P, Q, ε) .

also define any context information that is expected by the handlers (line 2) and that is provided by the client upon announcement (line 7).

Translucid contracts (Fig. 3.1-b, lines 3-10) allows one to establish the specification (**requires** in line 3 and **ensures** in line 10) that all handlers for this event must satisfy. It also allows to provide an abstract algorithm (**assumes** clause in lines 4-9) that all handlers must refine. Each handler's body (Fig. 3.1-b, lines 14-20) must match exactly this abstract algorithm, except for specification statements (line 8) that are refined by **refining** statement (lines 17-18) providing code (line 19) that implements the corresponding specification.

```
1  public event NotificationEvt{
2   Customer c;
3  }
4  public class ClientComponent {
5   void process(Customer c) {
6     ...
7     announce NotificationEvt(c){
8      c.notify()
9     };
10 }}
11 public class ProviderComponent {
12 void handler(NotificationEvt next){
13  ...
14  if (notificationEnabled){
15   next.invoke(); // proceed
16  }
17 }
18 when NotificationEvt do handler;
19
20 ProviderComponent(){
21  register(this);
22 }}
```

(a) basic

```
1  public event NotificationEvt{
2   Customer c;
3   requires true
4   assumes{
5    if (notificationEnabled){
6     next.invoke(); // proceed
7    }
8    requires true ensures c.ok();
9   }
10  ensures c.ok()
11 }
12 public class ProviderComponent {
13 void handler(NotificationEvt next){
14  if (notificationEnabled){
15   next.invoke(); // proceed
16  }
17  refining requires true
18          ensures c.ok(){
19    c.setOk();
20  };
21 }
22  ...
23 }}
```

(b) with translucid contract

Figure 3.1: Ptolemy event example

In AO programs the only advisable events correspond to the fixed set of *join point* types defined

by the language, like method calls or field accesses. In Ptolemy, on the contrary, the execution of any block of client code (the announced code) can be considered as announcing an event, giving more flexibility to the developer. From AO languages, Ptolemy takes the **`around`** advice and the **`proceed`** (**`invoke`** in Ptolemy) concepts. Every handler in Ptolemy corresponds to an advice that executes around the announced code. The **`invoke`** statement in the body of a handler corresponds to the AO **`proceed`** invocation.

Verification in Ptolemy is straightforward [7, 5, 55]. Every handler H for an event, like method *handler* in Fig. 3.1-b (lines 13-21), and also every piece of announced code S for that event, like the one in Fig. 3.1-a (line 8), must satisfy the specification, (P, Q), established in the translucid contract of the event (Fig. 3.1-b, lines 3 and 10): $H \sqsupseteq (P, Q)$ and $S \sqsupseteq (P, Q)$. In Ptolemy, the semantics of **`announce`** and **`invoke`** statements is that they execute a handler, H, or some announced code, S. Therefore, each of these instructions is verified as the non-deterministic choice between these two pieces of code. Considering that both of them satisfy the specification (P, Q) from the event, it follows that **`announce`** as well as **`invoke`** statements are verified using the specification $(P, Q)\Box(P, Q)$, that results in (P, Q). That is, $[announce\ E(..)\{S\}] \sqsupseteq (P, Q)$ and $[next.invoke()] \sqsupseteq (P, Q)$. As pointed out by the author in a previous work [55], imposing the same specification on both the handlers and the announced code is sound but has flexibility and completeness issues.

3.2 PtolemyRely: Separating Obligations of Subjects and Handlers

Using the same specification for both the subject (announced code) and the handlers has flexibility and completeness issues. An issue occurs if the business rules of a system require different behaviors from the subject and the handlers. Which specification must be put on the event declaration? If either of the two is selected then the program may not verify, since its counterpart would not

satisfy that specification. If the non-deterministic choice of both is used as the event specification, then the program will be valid, but the business rules may not be followed, as this specification is more general and may not guarantee the expected behaviors. Another issue occurs when the announced code has no effect (e.g., *skip*). As the event specification (P, Q) must be satisfied by the announced-code, then the Hoare triple $\{P\}skip\{Q\}$ must hold. From that it follows that $P \Rightarrow Q$ and then the handlers are limited to monotonic behaviors; i.e. ones that preserve the precondition.

In a previous work [55], the author presented an extension of the specification and reasoning features of Ptolemy, for dealing with the aforementioned issues. This extended language is named *PtolemyRely*. In PtolemyRely, each event type declares two specifications, one that must be satisfied by every handler, (P_H, Q_H), and one that must be satisfied by each piece of announced code, (P_B, Q_B).

```
1  public event NotificationEvt{
2   Customer c;
3   relies requires true ensures true // (P_B, Q_B)
4   requires true  // P_H
5   assumes{
6    if (notificationEnabled){
7     next.invoke(); // proceed
8    }
9    requires true ensures c.ok();
10  }
11  ensures c.ok() // Q_H
12 }
```

Figure 3.2: PtolemyRely event example

Handlers rely on the fact that the announced code satisfies (P_B, Q_B) and in turn they must guarantee to the client or base code that their execution satisfies (P_H, Q_H). **Announce** and **invoke** statements are reasoned about using the non-deterministic choice between these specifications: $[announce\ E(..)\{S\}] \sqsupseteq (P_H, Q_H) \Box (P_B, Q_B)$ and $[next.invoke()] \sqsupseteq (P_H, Q_H) \Box (P_B, Q_B)$. These features make PtolemyRely more flexible than Ptolemy and allows one to verify more programs.

Figure 3.2 illustrates these new features. The **relies** clause (line 3) introduces the specification (P_B, Q_B) for announced code. The specification (P_H, Q_H) surrounding the **assumes** clause (line 4 and 11) must be satisfied by every handler. Separate specifications allow one to verify each part of the program and respect the business rules. This also allows announced code that has no effect to be verified without limiting, in any way, the handlers' specification.

3.3 Static Verification of PtolemyRely Programs Using OpenJML

In another work [56] the author presented a mechanism for doing automated, modular static verification of PtolemyRely programs. The mechanism consists of translating PtolemyRely programs into JML (the Java Modelling Language) [37, 38, 13]. This allows one to use existing OpenJML [16] static verification tools to verify PtolemyRely programs. The translation is such that a PtolemyRely program is valid if and only if its encoding is a valid JML program.

JML [37, 38, 13] is a behavioral interface specification language tailored to Java. It allows one to specify the syntactic interface of Java modules (classes or interfaces) and its behavior from the client point of view. JML specifications include class invariants, method pre- and post-conditions and statement level assertions. Valid JML programs must satisfy the following conditions. The body, S, of every method, **requires** P **modifies** ε **ensures** Q $m()\{S\}$, must satisfy its specification, $S \sqsupseteq (P, Q, \varepsilon)$. Method calls are reasoned about modularly. At every method call its precondition is checked to hold and its postcondition is assumed afterwards. Every explicitly-stated assertion is verified.

OpenJML [16] is a set of tools for the JML language. It includes verifying (static checking) and run-time checking of JML programs. OpenJML static checking tools do automated deductive verification. They generate, using weakest precondition semantics, a proof script from the JML

specifications and the Java code and then use a SMT solver (like Yices [21] or Z3 [19]) for verification.

To verify a PtolemyRely program one performs several specific steps. Each event declaration includes a handler's specification (P_H, Q_H), a subject's specification (P_B, Q_B) and an abstract algorithm (**`assumes`** part) A. The body, H, of every handler should satisfy the handler's specification, $H \sqsupseteq (P_H, Q_H)$, and structurally refine the abstract algorithm, $H \sqsupseteq A$. **`Refining`** statements, **`refining requires`** P **`ensures`** Q $\{S\}$, require that its body refines its specification, $S \sqsupseteq (P, Q)$. **`Announce`** and **`invoke`** statements must be reasoned about using the non-deterministic choice specification, $(P_H, Q_H) \Box (P_B, Q_B)$. At event announcements of the form **`announce`** $E(..)\{B\}$, the announced-code must be verified against the corresponding specification in the event, $B \sqsupseteq (P_B, Q_B)$.

The referred work uses a JML class to simulate a *closure* construct for a block of code, S, and specification (P, Q). Figure 3.3, adapted from [56], shows the strategy. The basic idea is that a method, $execute$, is constructed. Its body corresponds to the block of code S and its specification corresponds to (P, Q). The frame $\overrightarrow{F}$ is computed from the free variables in P, S and Q.

The verification of this JML *closure* class certifies that the block of code satisfies the specification. The rest of the construction simulates the execution of the original block of code, with the added value of verifying its precondition and assuming its postcondition, according to the method call rules of JML. Using this $closure$ construct a PtolemyRely program, P, is translated into a JML program, *TR*$[\![P]\!]$, whose validity according to JML rules ensures the validity of the original program according to the rules of PtolemyRely.

```
closure   (P, Q, S) = {
          class Clr_s{
            public F⃗;  // free variables: free(S, P, Q)
            /*@ requires P; assignable F; @*/
            /*@ ensures Q; @*/
            public void execute(){S} // closure method
          }

          Clr_s clr_s = new Clr_s(); // instantiation
          clr_s.F⃗ = free(S, P, Q)⃗; // set
          clr_s.execute(); // execute
          free(S, P, Q)⃗ = clr_s.F⃗; // get
}
```

Figure 3.3: Closure construct. Here Clr_s is presumed to be a fresh class name for the closure class and F is the list of fields corresponding to the free variables in P, Q and S.

3.4 Methodology Oriented Approaches

Instead of relying mainly in language features, some approaches suggest attacking the problem of developing AO software from the methodological point of view. I discuss some of those approaches below.

Crosscut programming interfaces, or XPI, [59, 60] are at the core of the XPI-based design methodology. This methodology introduces a new design phase in which the crosscuting interfaces, mediating the base and aspect parts of the system, are designed before designing these parts. XPI decouple the advised base-code from the advising aspects. As an API defines a set of methods that will be implemented by one part and invoked by other part, an XPI defines a set of abstract join points that will be materialized by concrete join points in the base code and that will be advised by the corresponding aspects. In AspectJ, an XPI is defined by two auxiliary aspects: the *XPI* aspect and the *contract* aspect. The *XPI* aspect declares the signature of the pointcuts that comprise the interface, and a "partial" implementations of them, by providing their join point patterns. A

"real" aspect can use the pointcuts defined in this XPI aspect and bind them to the required advice. The *contract* aspect uses the pointcuts defined in the XPI aspect and advice them with code that checks that the informal contract is enforced. This informal contract establishes the precondition the base code has to ensure and the postcondition the advised code must guarantee. The lack of specific language support for XPI leaves in the developer the responsibility to adhere to the design discipline.

Cooperative aspect-oriented programming (Co-AOP) [32, 33] is a programming methodology based on the use of interfaces between the base and aspect code. Interfaces can be of different types. They can be just normal Java interfaces with static methods. The base code explicitly invoke the interface methods and the aspect matches and advices these calls. Interfaces can also be XPI's defining pointcuts in an auxiliary aspect. They obliviously pick join points in the base code and can be advised by provider aspect that match the XPI pointcuts. The favourite type of interfaces in Co-AOP are the so-called explicit join points (EJP) [31]. EJP are similar to method declarations in an interface. They are defined inside a convenience aspect that acts as an interface. EJP's are explicitly referenced in the base code and they can advise arbitrary blocks of base code, similar to the `announce` statement in Ptolemy. Nevertheless EJP does not explicitly support specification and verification features, like Ptolemy does.

CHAPTER 4: REASONING TRADEOFFS

When reasoning about object oriented, implicit invocation and aspect oriented programs a series or decisions or tradeoffs should be considered. These decisions will affect important properties of the reasoning process itself, like its completeness and precision, its easiness and its applicability to practical software engineering situations.

4.1 Modular vs. Non-Modular Reasoning

Modular reasoning is understood as being able to establish properties of a module just by considering its interface, specification and implementation; and the interfaces and specifications, not the implementation, of modules referenced by them. The specification information for each module includes the assumptions it makes (preconditions) and the guarantee (postcondition) it promises [17]. For doing modular reasoning it is required that each module has a specification and also that the references in the invoking module allows one to identify the specifications to use for reasoning about the invocations.

Modular reasoning can be formalized as follows. A *client* module s has a specification $(P_s, Q_s, \varepsilon_s)$ and a body b_s. In the body b_s there are references (invocations) to *provider* modules $m_1, \ldots, m_n$, each of which having its corresponding specification $(P_i, Q_i, \varepsilon_i)$ and body b_i. To modularly reason about s demands that its body b_s, when composed with the specifications $(P_i, Q_i, \varepsilon_i)$ satisfies s's own specification: $b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}] \sqsupseteq (P_s, Q_s, \varepsilon_s)$. The composition $b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}]$ represents the program b_s with every invocation of any of the modules m_i replaced by the specification statement "**`requires`** P_i **`modifies`** ε_i **`ensures`** Q_i", corresponding to the specification $(P_i, Q_i, \varepsilon_i)$ of module m_i. For modular reasoning to be sound, every module m_i must be verified

in the same way to satisfy its own specification, $(P_i, Q_i, \varepsilon_i)$.

There are various circumstances under which modular reasoning cannot be applied directly. If there are no specifications for the invoked modules then their bodies should be used instead, preventing modular reasoning. Unless there is recursion, a non-modular approach can be used, the client body, composed with the bodies of the invoked modules, is checked against its expected behavior: $b_s \circledcirc [(b_i)_{i:1..n}] \sqsupseteq (P_s, Q_s, \varepsilon_s)$. In spite of the many practical advantages of modular reasoning, this non-modular reasoning is at least as precise as modular reasoning. That is due to the fact that the body of the client module, b_s, composed with the bodies of the invoked modules, b_i, refines that same client body, b_s, composed with the specifications, $(P_i, Q_i, \varepsilon_i)$, for those invoked modules: $(b_s \circledcirc [(b_i)_{i:1..n}]) \sqsupseteq (b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}])$. This follows from the condition that $b_i \sqsupseteq (P_i, Q_i, \varepsilon_i)$ and from standard results in refinement calculus [3, 45], regarding the monotonicity of programs with respect to refinement of its parts, as also used in [57]. Every specification, $(P_s, Q_s, \varepsilon_s)$, satisfied by the later program is also satisfied by the former one:
$[(b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}]) \sqsupseteq (P_s, Q_s, \varepsilon_s)] \Rightarrow [(b_s \circledcirc [(b_i)_{i:1..n}]) \sqsupseteq (P_s, Q_s, \varepsilon_s)]$, but there could be specifications satisfied by the former and not satisfied by the later, making the non-modular reasoning at least as precise as the modular one.

If the invoked module cannot be uniquely identified at an invocation then the particular specification to reason about this invocation will not be known, again preventing modular reasoning. This happens in the case of object oriented dynamic dispatching and also in the case of implicit invocation, both explained further below. In the case of aspect orientation the client module does not invoke any advice at all, instead the pointcuts designators in the aspects select the join points where to apply the advice. In this case no modular reasoning is possible, unless obliviousness is relaxed in some way.

In *static dispatching* each invocation is dispatched to the only one corresponding method imple-

mentation in the static type of the target object. In this case modular reasoning can be done exactly as described above.

In the case of object oriented *dynamic dispatching*, as described in Section 2.5, each method invocation can be dispatched to the corresponding method implementation in any of the subtypes of the target's static type, including itself. The exact module to be invoked at runtime is not known, and so the particular specification to reason about this invocation is also unknown. In this case, modular reasoning can be recovered by *supertype abstraction*, using the method's specification in the target's static type to reason about the invocation. Client reasoning can be done as in the case of static dispatching, reasoning each method invocation $e_i.m_i()$ using the specification given to that method in the static type of the target object. The notation $(P^T_m, Q^T_m, \varepsilon^T_m)$ represents the specification given to a method m in a type T; and T_i is the static type of the target object e_i. $b_s \circledcirc [(P^{T_i}_{m_i}, Q^{T_i}_{m_i}, \varepsilon^{T_i}_{m_i})_{i:1..n}] \sqsupseteq (P_s, Q_s, \varepsilon_s)$. For supertype abstraction to be sound, modules must adhere to the behavioral subtyping discipline, which requires that the specification for a method in a subtype refines the specifications for that method in the supertypes [40]. The provider modules not only must be checked against their corresponding specifications as before, but also to satisfy the behavioral subtyping property.
$(\forall T \bullet T \leq T_i \Rightarrow (P^T_{m_i}, Q^T_{m_i}, \varepsilon^T_{m_i}) \sqsupseteq (P_i, Q_i, \varepsilon_i))$, where $(P_i, Q_i, \varepsilon_i)$ is the specification for method m_i in its declaring type T_i, that is $(P_i, Q_i, \varepsilon_i) = (P^{T_i}_{m_i}, Q^{T_i}_{m_i}, \varepsilon^{T_i}_{m_i})$.

In *Implicit Invocation* languages, instead of explicit invocations of specific modules, events are announced and the runtime system implicitly invokes the registered *handlers* for the event. Not only are the dispatched methods unknown, like in dynamic dispatching, but many of them may be executed, in contrast to dynamic dispatching where exactly one is executed. Along the lines of supertype abstraction, modular reasoning can be recovered in implicit invocation using *handler abstraction*. In this work the term handler abstraction is coined to mean that for each event, e, a handlers' specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, can be defined which must be satisfied by every handler,

h_{e_i}, for that event.

In *full delivery* implicit invocation [10] all the handlers for an event are invoked by the system in some arbitrary order [26]. Examples of this scheme are traditional implicit invocation systems and AO **`before`** and **`after`** advice. Full delivery requires that the execution of any handler leaves the system in a state in which any other handler can be executed. Therefore, the postcondition for the handlers must imply their precondition, $Q_{e_H} \Rightarrow P_{e_H}$, and so this precondition, P_{e_H}, becomes an invariant that must be kept by the handlers. Under this premise the client module can be modularly reasoned about, using the corresponding event's handlers specification to reason about each event announcement, and considering the cases where there are no registered handlers as **`skip`**.

In *single delivery* implicit invocation only one handler is invoked in response to an event, and it depends on that handler whether the next handler is invoked. The same applies to the second handler and so on. Examples of single delivery are AO **`around`** advice and Ptolemy events. In AO a **`proceed`** instruction in the body of a piece of advice (handler) invokes the next handler. Similarly, in Ptolemy an **`invoke`** statement invokes the next handler. In this case modular reasoning is recovered by checking each handler against the handler's specification for the corresponding event. Handlers are not required to satisfy an invariant, as in the case of full delivery. Instead, in the reasoning of each handler, **`proceed`** or **`invoke`** statements need to be considered, using the handler's specification to substitute them: $h_{e_i} \circledcirc [(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. Again special considerations must be taken when there are no more handlers to execute.

More details about the modular vs. non-modular reasoning will be analysed in Chapter 5.

4.2 Case analysis vs. Abstraction

There are situations, like dynamic dispatching and implicit invocation, where an invocation (method call or event announcement) can be dispatched to different provider modules, each one with its own specification, $(P_i, Q_i, \varepsilon_i)$. In this situations it is not clear what specification to use for reasoning about the invocation. One option is to use abstraction and another one is to use case-by-case reasoning.

The abstraction approach consists in computing a specification, (P, Q, ε), that abstracts all the specifications for the provider modules, $(P, Q, \varepsilon) \sqsupseteq (P_i, Q_i, \varepsilon_i)_{i:1..n}$, and using it to reason about the invocation, $(call \backslash announce) \sqsupseteq (P, Q, \varepsilon)$. This approach admits modular reasoning.

On the contrary, case analysis considers the reasoning requirements for each case separately, instead of abstracting them all in a general specification. Abstraction mechanisms, like supertype abstraction, are sound and modular but not complete, since valid programs cannot be reasoned about using them. The example in Figure 4.1, adapted from [20], illustrates this situation.

```
1  class A{
2    int x; int y;
3    /*@ ensures x=\old(x)+1
4          && y=\old(y)+1; @*/
5    void inc(){x=x+1;y=y+1;}
6
7    //@ ensures x=\old(x)+2;
8    void incX2(){inc();inc();}
9  }
```

```
10 class Ax extends A{
11   /*@ ensures x=\old(x)+1;@*/
12   void inc(){x=x+1;}
13 }
```

Figure 4.1: Supertype Abstraction

The example presents a case of supertype abstraction, without specification inheritance. Despite being valid, the program in Figure 4.1 does not follow the behavioral subtyping discipline. Class `Ax` is not a behavioral subtype of class `A`, because the specification for method `inc()` in `Ax` does not refine its specification in `A`.

The program is valid for all the methods in it satisfy their specifications, even in the presence of dynamic dispatching. Method `inc()` satisfies its corresponding specifications in both classes `A` and `Ax`. Method `incX2()` is valid since the *requirements* it imposes on method `inc()` (that it at least increments variable `x`) are part of the *commitments* of this method in both classes `A` and `Ax`, despite `Ax` not being a behavioral subtype of `A`. This relaxed discipline has been termed *lazy behavioral subtyping*[20]. It can be seen as case analysis, since each class is analysed as a separate case. The requirements some of its methods impose on others are verified against the commitments of these other methods, plus their inherited commitments.

4.3 Explicit Invocation vs Implicit Invocation

In explicit invocation the client component explicitly calls methods from the provider component. Explicit invocation eases debugging and reasoning. In the case of static dispatching, invocations can be reasoned about using the specification of the invoked methods. In the case of dynamic dispatching, the behavioral subtyping discipline can be adopted, enabling the use of supertype abstraction for reasoning the invocations.

With implicit invocation, the client component announces events and the system implicitly invokes the target components that have been registered for these events. Implicit invocation has many advantages. The loosely coupling between components eases system evolution and reuse. Target components can be added, changed or removed without modifying the client. Independent devolp-

ment is also facilitated. There are also disadvantages. Reasoning about the client is more involved. There are no references to the target components that will be invoked, nor the number of them or the order of execution is known. As pointed out in Section 4.1, modular reasoning can be recovered using handler abstraction, while considering the both single and full delivery.

4.4 Explicit Announcement vs Implicit Announcement

In a general sense, explicit announcement means that by inspecting the client code one can determine the places where an announcement is made, signalling that certain functionality would be invoked at those places. In traditional object oriented programs the announcements correspond to method invocations, that are explicitly made in the client code. In event-based systems the language provides some type of `announce` construct that is used to explicitly announce events, which will cause the implicit invocation of handler methods from the provider components. The main advantage of explicit announcement is that it provides information for reasoning about the client code. Explicitly knowing where the event announcements are made, what is left is to reason about them, as was described in Section 4.3.

With implicit announcement, as in aspect oriented programming, the client remains oblivious [25] or ignorant of where and what functionality (advice) may be invoked. Different events occurring at certain join points during the execution of the client code, like method calls or field accesses, are considered candidates that would cause the invocation of handlers (advice). Pointcuts inside aspects select or quantify [25] the event-announcing points in the client code, where advice from the aspect is added. Reasoning about the client code requires to identify the announcing points, selected by the pointcuts, and then reason about this announcements. This reasoning should consider the effect of the advised join point and the effect of the advice added. The main advantage of implicit announcement is that it allows developers to add or alter the functionality of the client

code without changing it all. This is particularly useful for implementing cross-cutting concerns. A cross-cutting functionality can be encapsulated in only one place, in an aspect component, and applied in many places of a system without changing it. However this obliviousness is also a severe disadvantage, particularly for reasoning the client code. There is no clue in it about where and what functionality may be added.

One simple solution proposed [36] to reason about AO system with implicit announcement and implicit invocation is to do it in two phases. A whole program non-modular analysis first determine the advised points and then a modular reasoning is applied to each one of them. This second phase falls back to reason an implicit invocation system. Further explanation is given in section 5.5.

CHAPTER 5: REASONING SCENARIOS AND THEIR PROOF RULES

As previously stated, the problem I aim to solve in this work is to show how to do formal reasoning for specific scenarios in implicit invocation and aspect oriented languages (based on Java) and to provide guidance to language designers and developers on methodological aspects of their use. Before detailing the scenarios, some definitions and results are needed.

5.1 Algebra of Specifications

Definition 2. *(One-State Predicate Valuation) The valuation of a predicate P in a state α is defined as the predicate with every free variable substituted by its corresponding value in α: $[\![P]\!]_\alpha \equiv P[\overrightarrow{\alpha(x)}/\overrightarrow{x}]$, where $\overrightarrow{x} = FV(P)$ are the free variables of P.*

Definition 3. *(Two-State Predicate Valuation) The valuation of a two-state predicate P in two states α_1, α_2 is defined as the predicate with every* **`old`** *free variable substituted by its corresponding value in state α_1 and every free normal variable substituted by its corresponding value in state α_2: $[\![P]\!]_{\alpha_1,\alpha_2} \equiv P[\overrightarrow{\alpha_1(x)}/\overrightarrow{old(x)},\ \ \overrightarrow{\alpha_2(y)}/\overrightarrow{y}]$, where $\overrightarrow{old(x)} = OFV(P)$ are the* **`old`** *free variables of P and $\overrightarrow{y} = FV(P)$ are the normal free variables of P.*

In a Hoare formula of the form $\{P\}S\{Q\}[\varepsilon]$ the precondition P is an one-state predicate and the postcondition Q is a two-state predicate. The formula is valid if whenever the program S starts execution in a state α_1 such that $[\![P]\!]_{\alpha_1}$ holds and terminates in a state α_2 then $[\![Q]\!]_{\alpha_1,\alpha_2}$ holds; and the program only modifies locations in ε.

Theorem 4 (Specification Refinement)**.** *The refinement relation ($\sqsupseteq$) between two specifications can be characterized as follows, where $old(P) = P[\overrightarrow{old(x)}/\overrightarrow{x}]$ for $\overrightarrow{x} = FV(P)$:*

$$\left[\Big((P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon)\Big) \Leftrightarrow \Big((P \Rightarrow P') \land ((old(P) \land Q') \Rightarrow Q) \land (\varepsilon' \subseteq \varepsilon)\Big)\right] \qquad (4.1)$$

Proof: [Specification Refinement]

$\langle$by definition of refinement (1) it is required to prove that$\rangle$

$$\left[\begin{array}{ll} & \Big(\forall\, program\ S,\ (S \sqsupseteq (P', Q', \varepsilon')) \Rightarrow (S \sqsupseteq (P, Q, \varepsilon))\Big) \\ \Longleftrightarrow & \Big((P \Rightarrow P') \land ((old(P) \land Q') \Rightarrow Q) \land (\varepsilon' \subseteq \varepsilon)\Big) \end{array}\right]$$

$\langle$by adoption of proof of proposition 10 from [39]$\rangle$

$$\left[\Big((P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon)\Big) \Leftrightarrow \Big((P \Rightarrow P') \land ((old(P) \land Q') \Rightarrow Q) \land (\varepsilon' \subseteq \varepsilon)\Big)\right] \quad \blacksquare$$

For example $(x \geq 0,\ x \geq old(x) + 1,\ \{x\}) \sqsupseteq (x > 0,\ x > 1,\ \{x\})$, as $(x > 0) \Rightarrow (x \geq 0)$ and $(\ (old(x) > 0) \land (x \geq old(x) + 1\ ) \Rightarrow (x > 1)$.

Lemma 5 (Meet or greatest lower bound of specifications w.r.t. refinement)**.**
$\sqcap_{i=1...n}(P_i, Q_i, \varepsilon_i) = (\land_{i=1...n}P_i, \lor_{i=1...n}Q_i, \cup_{i=1...n}\varepsilon_i)$*, or in the binary case:*
$(P_j, Q_j, \varepsilon_j) \sqcap (P_k, Q_k, \varepsilon_k) = (P, Q, \varepsilon) = (P_j \land P_k,\ Q_j \lor Q_k,\ \varepsilon_j \cup \varepsilon_k)$*, that is*

$$\left[\begin{array}{c} \Big((P_j, Q_j, \varepsilon_j) \sqsupseteq (P, Q, \varepsilon)\Big) \land \qquad (5.1) \\ \Big((P_k, Q_k, \varepsilon_k) \sqsupseteq (P, Q, \varepsilon)\Big) \land \qquad (5.2) \\ \Big([(P_j, Q_j, \varepsilon_j) \sqsupseteq (P', Q', \varepsilon') \land (P_k, Q_k, \varepsilon_k) \sqsupseteq (P', Q', \varepsilon')] \qquad (5.3) \\ \Rightarrow (P, Q, \varepsilon) \sqsupseteq (P', Q', \varepsilon')\Big) \end{array}\right]$$

Proof: [Meet of specifications]

5\. 1. $\langle$by predicate calculus and set theory$\rangle$

$$\left[\Big((P_j \land P_k) \Rightarrow P_j\Big)\ \land\ \Big((old(P_j) \land Q_j) \Rightarrow (Q_j \lor Q_k)\Big)\ \land\ \Big(\varepsilon_j \subseteq (\varepsilon_j \cup \varepsilon_k)\Big)\right]$$

$\Longrightarrow$ $\langle$by theorem 4$\rangle$

$$\left[(P_j, Q_j, \varepsilon_j) \sqsupseteq (P_j \wedge P_k,\ Q_j \vee Q_k,\ \varepsilon_j \cup \varepsilon_k)\right]$$

$\Longrightarrow$ ⟨by definition of (P, Q, ε)⟩

$$\left[(P_j, Q_j, \varepsilon_j) \sqsupseteq (P, Q, \varepsilon)\right] \blacksquare$$

5. 2. ⟨by similar to previous one⟩

$$\left[(P_k, Q_k, \varepsilon_k) \sqsupseteq (P, Q, \varepsilon)\right] \blacksquare$$

5. 3. ⟨by theorem 4⟩

$$\left[\begin{array}{c} \Big((P' \Rightarrow P_j) \wedge ((old(P') \wedge Q_j) \Rightarrow Q') \wedge (\varepsilon_j \subseteq \varepsilon')\Big) \wedge \\ \Big((P' \Rightarrow P_k) \wedge ((old(P') \wedge Q_k) \Rightarrow Q') \wedge (\varepsilon_k \subseteq \varepsilon')\Big) \end{array}\right]$$

$\Longrightarrow$ ⟨by predicate calculus and set theory⟩

$$\left[\Big(P' \Rightarrow (P_j \wedge P_k)\Big) \wedge \Big((old(P') \wedge (Q_j \vee Q_k)) \Rightarrow Q'\Big) \wedge \Big((\varepsilon_j \cup \varepsilon_k) \subseteq \varepsilon'\Big)\right]$$

$\Longrightarrow$ ⟨by definition of (P, Q, ε)⟩

$$\left[\Big(P' \Rightarrow P\Big) \wedge \Big((old(P') \wedge Q) \Rightarrow Q'\Big) \wedge \Big(\varepsilon \subseteq \varepsilon'\Big)\right]$$

$\Longrightarrow$ ⟨by theorem 4⟩

$$\left[(P, Q, \varepsilon) \sqsupseteq (P', Q', \varepsilon')\right] \blacksquare$$

Lemma 6 (Join or least upper bound of specifications w.r.t. refinement)**.**

$\sqcup_{i=1\ldots n}(P_i, Q_i, \varepsilon_i) = (\vee_{i=1\ldots n}P_i, \wedge_{i=1\ldots n}(old(P_i) \Rightarrow Q_i), \cap_{i=1\ldots n}\varepsilon_i)$*, or in the binary case:*

$(P_j, Q_j, \varepsilon_j) \sqcup (P_k, Q_k, \varepsilon_k) = (P, Q, \varepsilon) = (P_j \vee P_k,\ (old(P_j) \Rightarrow Q_j) \wedge (old(P_k) \Rightarrow Q_k),\ \varepsilon_j \cap \varepsilon_k)$*,*

that is

$$\left[\begin{array}{c} \Big((P, Q, \varepsilon) \sqsupseteq (P_j, Q_j, \varepsilon_j)\Big) \wedge \quad (6.1) \\ \Big((P, Q, \varepsilon) \sqsupseteq (P_k, Q_k, \varepsilon_k)\Big) \wedge \quad (6.2) \\ \Big([((P', Q', \varepsilon') \sqsupseteq (P_j, Q_j, \varepsilon_j)) \wedge ((P', Q', \varepsilon') \sqsupseteq (P_k, Q_k, \varepsilon_k))] \quad (6.3) \\ \Rightarrow [(P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon)]\Big) \end{array}\right]$$

Proof: [Join of specifications]

6. 1. $\langle$by predicate calculus and set theory$\rangle$

$$\left[\begin{array}{c}\Big(P_j \Rightarrow (P_j \vee P_k)\Big) \ \wedge \\ \Big([old(P_j) \wedge ((old(P_j) \Rightarrow Q_j) \wedge (old(P_k) \Rightarrow Q_k))] \Rightarrow Q_j\Big) \ \wedge \\ \Big((\ \varepsilon_j \cap \varepsilon_k) \subseteq \varepsilon_j\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by theorem 4$\rangle$

$$\Big[(P_j \vee P_k,\ (old(P_j) \Rightarrow Q_j) \wedge (old(P_k) \Rightarrow Q_k),\ \varepsilon_j \cap \varepsilon_k) \sqsupseteq (P_j, Q_j, \varepsilon_j)\Big]$$

$\Longrightarrow$ $\langle$by definition of $(P, Q, \varepsilon)$$\rangle$

$$\Big[(P, Q, \varepsilon) \sqsupseteq (P_j, Q_j, \varepsilon_j)\Big] \ \blacksquare$$

6. 2. $\langle$by similar to previous one.$\rangle$

$$\Big[(P, Q, \varepsilon) \sqsupseteq (P_k, Q_k, \varepsilon_k)\Big] \ \blacksquare$$

6. 3. according to theorem 4 it is required to prove that

$$\left[\begin{array}{c}\Big((P_j \Rightarrow P') \wedge ((old(P_j) \wedge Q') \Rightarrow Q_j) \wedge (\varepsilon' \subseteq \varepsilon_j)\Big) \ \wedge \\ \Big((P_k \Rightarrow P') \wedge ((old(P_k) \wedge Q') \Rightarrow Q_k) \wedge (\varepsilon' \subseteq \varepsilon_k)\Big)\end{array}\right]$$

$\Longrightarrow$

$$\left[\begin{array}{c}\Big((P_j \vee P_k) \Rightarrow P'\Big) \ \wedge \\ \Big(\varepsilon' \subseteq (\varepsilon_j \cap \varepsilon_k)\Big) \ \wedge \\ \Big((old(P_j \vee P_k) \wedge Q') \Rightarrow ((old(P_j) \Rightarrow Q_j) \wedge (old(P_k) \Rightarrow Q_k))\Big)\end{array}\right]$$

$\langle$by hypothesis$\rangle$

$$\left[\begin{array}{c}\Big((P_j \Rightarrow P') \wedge ((old(P_j) \wedge Q') \Rightarrow Q_j) \wedge (\varepsilon' \subseteq \varepsilon_j)\Big) \ \wedge \\ \Big((P_k \Rightarrow P') \wedge ((old(P_k) \wedge Q') \Rightarrow Q_k) \wedge (\varepsilon' \subseteq \varepsilon_k)\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by predicate calculus and set theory$\rangle$

$$\Big[\Big((P_j \vee P_k) \Rightarrow P'\Big) \ \wedge \ \Big((\varepsilon_j \cup \varepsilon_k) \subseteq \varepsilon'\Big)\Big]$$

$\Longrightarrow$ $\langle$by Q' and $((old(P_j) \wedge Q') \Rightarrow Q_j)$$\rangle$

$$\left[old(P_j) \Rightarrow Q_j\right]$$

$\Longrightarrow$ ⟨by Q' and $((old(P_k) \wedge Q') \Rightarrow Q_k)$⟩

$$\left[old(P_k) \Rightarrow Q_k\right] \blacksquare$$

Definition 7. *(Non-Deterministic Choice) The non-deterministic choice ($\Box$) between two specifications is defined as:* $(P, Q, \varepsilon)\Box(P', Q', \varepsilon') \equiv (P \wedge P',\ Q \vee Q',\ \varepsilon \cup \varepsilon') = (P, Q, \varepsilon) \sqcap (P', Q', \varepsilon')$.

Lemma 8 (choice absorption w.r.t. refinement)**.**

$$\left[\Big((P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon)\Big) \iff \Big(((P', Q', \varepsilon')\Box(P, Q, \varepsilon)) \sqsupseteq (P, Q, \varepsilon)\Big)\right] \tag{8.1}$$

Proof: [choice absorption w.r.t. refinement]

⟨by starting from the left hand side⟩

$$\left[(P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon)\right]$$

$\iff$ ⟨by theorem 4⟩

$$\left[(P \Rightarrow P') \wedge ((old(P) \wedge Q') \Rightarrow Q) \wedge (\varepsilon' \subseteq \varepsilon)\right]$$

$\iff$ ⟨by predicate calculus and set theory⟩

$$\left[((P \Rightarrow P') \wedge true) \wedge (((old(P) \wedge Q') \Rightarrow Q) \wedge true) \wedge ((\varepsilon' \cup \varepsilon) \subseteq \varepsilon)\right]$$

$\iff$ ⟨by predicate calculus⟩

$$\left[((P \Rightarrow P') \wedge (P \Rightarrow P)) \wedge (((old(P) \wedge Q') \Rightarrow Q) \wedge ((old(P) \wedge Q) \Rightarrow Q)) \wedge ((\varepsilon \cup \varepsilon') \subseteq \varepsilon)\right]$$

$\iff$ ⟨by predicate calculus⟩

$$\left[(P \Rightarrow (P' \wedge P)) \wedge ((old(P) \wedge (Q' \vee Q)) \Rightarrow Q) \wedge ((\varepsilon' \cup \varepsilon) \subseteq \varepsilon)\right]$$

$\iff$ ⟨by theorem 4⟩

$$\left[(P' \wedge P,\ Q' \vee Q,\ \varepsilon' \cup \varepsilon) \sqsupseteq (P, Q, \varepsilon)\right]$$

$\iff$ ⟨by definition of choice $\Box$⟩

$$\left[((P', Q', \varepsilon')\Box(P, Q, \varepsilon)) \sqsupseteq (P, Q, \varepsilon)\right] \blacksquare$$

Lemma 9 (Meet and Join choice)**.**

$$\left[\left(\sqcap_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\ \Box\ \sqcup_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\sqsupseteq\left(\sqcap_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\right] \tag{9.1}$$

Proof: [Meet and Join choice]

⟨by definition of Join and Meet⟩

$$\left[\left(\sqcup_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\ \sqsupseteq\ \left(\sqcap_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\right]$$

⟨by lemma 8⟩

$$\left[\left(\sqcup_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\ \Box\ \sqcap_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\sqsupseteq\left(\sqcap_{i=1\ldots n}(P_i,Q_i,\varepsilon_i)\right)\right]\ \blacksquare$$

5.2 The Reference Scenario

The *Reference Scenario* is a very general setting in which the different scenarios are configured. A *client* module $s()$ has a specification (P_s,Q_s,ε_s) and a body b_s. In the body b_s there are blocks of code $\{B_i\}$ at which provider modules could be invoked.

```
//@ requires Ps;
//@ modifies εs;
//@ ensures Qs;
s(){
 ...
 {B1}
 ...
 {Bj}
 ...
 {Bn}
 ...
}
```

Figure 5.1: Reference Scenario

Each block should be reasoned about to satisfy a corresponding specification.

$$\{B_i\} \sqsupseteq (P_i, Q_i, \varepsilon_i) \tag{5.1}$$

The body of the client, considering the reasoning of the blocks, should satisfy its specification.

$$b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}] \sqsupseteq (P_s, Q_s, \varepsilon_s) \tag{5.2}$$

The composition $b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}]$ represents the program b_s with each block $\{B_i\}$ substituted by its corresponding specification $(P_i, Q_i, \varepsilon_i)$.

5.3 The Object Oriented Scenario

As illustrated in Figure 5.2, in the *Object Oriented* scenario the blocks $\{B_i\}$ of the general scenario (Fig. 5.1) correspond to method invocations, $e_i.m_i()$. This is a case of explicit announcement and explicit invocation. The method-calls explicitly announce that some functionality will be invoked and the name used in the call explicitly indicates what method will be invoked.

```
//@ requires Ps;
//@ modifies εs;
//@ ensures Qs;
s(){
 ...
 e1.m1(...);
 ...
 ej.mj(...);
 ...
 en.mn(...);
 ...
}
```

Figure 5.2: OO Scenario

The blocks can be reasoned about using the specification of the corresponding invoked methods. Modularity can be achieved using supertype abstraction supported by behavioral subtyping. As is standard [40, 51, 34], for reasoning a method invocation, $e.m()$, the proof rule in (5.3) uses the specification, $(P^T_m, Q^T_m, \varepsilon^T_m)$, for the invoked method, m, in the static type, T, of the target object, e. This specification is given by the function $specOf()$. The function $declOf()$ correspondingly gives the method declaration. Both functions return the most specific information for a method, m, in the type hierarchy from the root type (object) down to a given type, T.

(OO-INVOKE$_R$)

$$\frac{\begin{array}{c}\Gamma(e) = T, \qquad specOf(T, m) = (P^T_m, Q^T_m, \varepsilon^T_m), \\ declOf(T, m) = T_m\ m(T_1 var_1, \ldots, T_n var_n)\end{array}}{\{P^T_m[e/this, e_i/var_i]\}\ e.m(e_1, \ldots, e_n)\ \{Q^T_m[e/this, e_i/var_i]\}[\varepsilon^T_m[e/this, e_i/var_i]]} \quad (5.3)$$

For this rule to be sound the program being reasoned about must adhere to behavioral subtyping principle [50], which requires that the specification for a method in a subtype refines the specifications for that method in the supertypes [40].

Given a type T and a method m in T, behavioral subtyping requires that

$$\forall S \cdot S \leq T \Rightarrow (P^S_m, Q^S_m, \varepsilon^S_m) \sqsupseteq (P^T_m, Q^T_m, \varepsilon^T_m) \quad (5.4)$$

As mentioned in Section 4.2, this discipline is sound but incomplete, as there exist correct programs that cannot be verified using it.

5.4 II/EA Scenarios

In the II/EA scenarios the blocks $\{B_i\}$ of the general scenario (Fig. 5.1) correspond to event announcements. An **announce** construct explicitly announces (*EA*) an event and the runtime system *delivers* the event to the provider components, by implicitly invoking (*II*) the handlers registered for that event. At event announcement, context information can be passed to the handlers. There are two modes of delivery, full and single, that must be considered.

5.4.1 II/EA Full Delivery Scenario

In full delivery mode, the announced event is broadcast to *all* the corresponding handlers, which are invoked in some arbitrary order. The client announces events as illustrated in Figure 5.3.

```
//@ requires Ps;
//@ modifies εs;
//@ ensures Qs;
s(){
 ...
 announce e1(...);
 ...
 announce ej(...);
 ...
 announce en(...);
 ...
}
```

Figure 5.3: II/EA Full Delivery Mode

For a given announcement of an event, e, the runtime system invokes all the register handlers, h_i, for that event, as illustrated in Figure 5.4. As pointed out in Section 4.1, for doing modular reasoning in implicit invocation languages *handler abstraction* can be used. For each event, e, a

handlers' specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, is defined that must be satisfied by its handlers, $h_{i=1...m}$.

$$\forall_{i=1...m} \cdot h_i \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \tag{5.5}$$

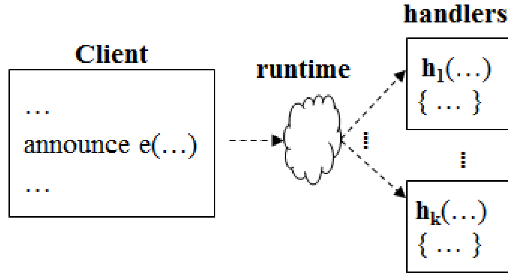

Figure 5.4: Full Delivery at Runtime

In full delivery, the execution of any handler should leave the system in a state in which other handlers can be executed. Therefore, the precondition, P_{e_H}, becomes an invariant that must be kept by the handlers. That means that the postcondition can be chosen the same as the precondition:

$$Q_{e_H} \equiv P_{e_H} \tag{5.6}$$

To reason about the announcement of an event there are two cases that must be considered, when there are registered handlers and when there are not. When there are registered handlers the announcement can be reasoned about using the handlers' specification, $(P_{e_H}, P_{e_H}, \varepsilon_{e_H})$. When there are no registered handlers, the announcement behaves like **skip**, and can be reasoned about by $(P_{e_H}, P_{e_H}, \{\})$. The non-deterministic choice between them can be easily computed, and proof rule 5.7 builds on it.

(II-FD-ANNOUNCE$_R$)

$$\frac{(\textbf{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT, \quad contract_H = \textbf{invariant}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}}{CT, \Gamma \vdash \{P_{e_H}[e_i/var_i]\}\ \textbf{announce}\ e(e_1, \ldots, e_n)\ \{P_{e_H}[e_i/var_i]\}[\varepsilon_{e_H}[e_i/var_i]]} \tag{5.7}$$

5.4.2 II/EA Single Delivery Scenarios

The idea in this scenario is that certain blocks of code of a program are considered to trigger events. The execution of any of these blocks is postponed. Instead a handler for the corresponding event is invoked and the triggering block is "passed" to it. It depends on the handler to proceed to execute the original block or not. If there are no registered handlers for the event only the block is executed.

In II/EA single delivery mode an **announce** construct announces an event and, along with context information, passes the announced block of code, as shown in Figure 5.5(b).

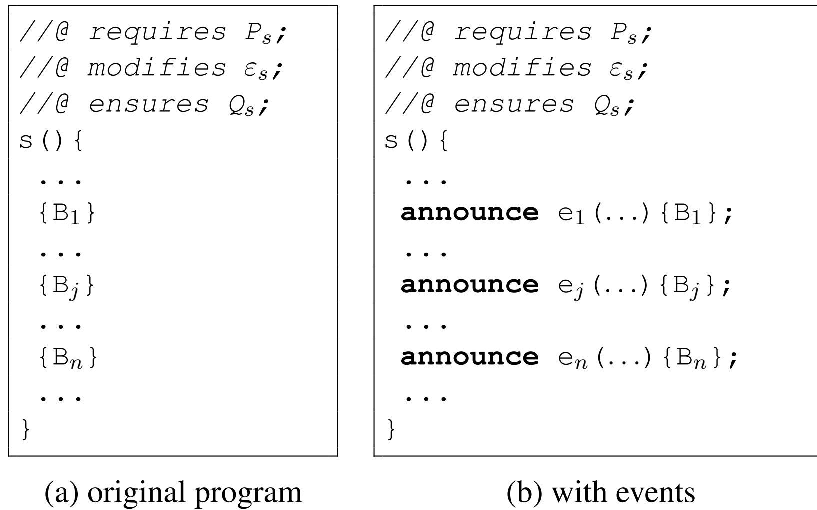

Figure 5.5: II/EA Single Delivery Mode

In the body of a handler, **invoke** statements invoke the next handler if any or the announced code otherwise. This is illustrated in Figure 5.6.

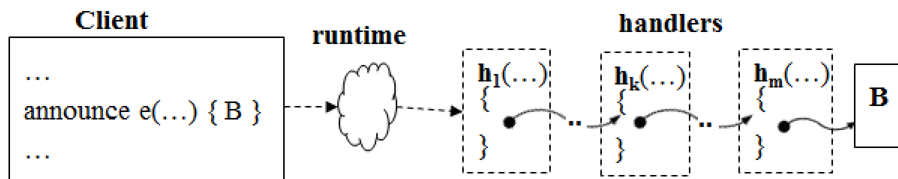

Figure 5.6: II/EA Single Delivery at Runtime

The reasoning of an **announce** statement should consider the specification for any handler, in case there are registered handlers, and the specification of the particular announced block of code, in case there are no registered handlers. This particular block of code is known because it is part of the **announce** statement that is being reasoned about. The reasoning of an **invoke** statement in the body of a handler should consider the specification for any next handler and for any announced block of code. In this case neither the next handler nor the block of code is known, as the **invoke** statement could be in the execution chain for any announcement for the corresponding event. For doing modular reasoning in II/EA single delivery scenarios it is required an specification that abstracts all the handlers for an event (*handler abstraction*) and a specification that abstracts all the triggering blocks of code for that event (*trigger abstraction*). Those are the handler's specification $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$ and the base-code specification $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, as depicted in Figure 5.7.

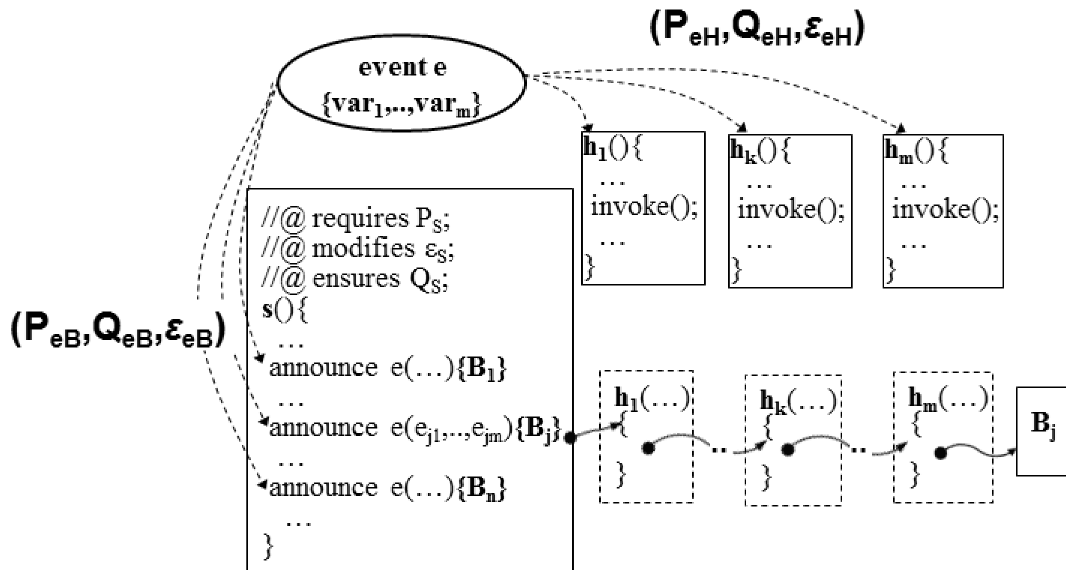

Figure 5.7: II/EA Single Delivery Abstraction

The behavior of each triggering block of code B_i in the original program is characterized by a corresponding specification $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})$. *Trigger abstraction* requires to choose a base-code

specification $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$ that generalizes all the block's specifications:

$$\forall_{i=1...n} \bullet \Big(B_i \sqsupseteq (P_{B_i}, Q_{B_i}, \varepsilon_{B_i})_{[e_{i_k}/var_k]}\Big) \wedge \Big((P_{B_i}, Q_{B_i}, \varepsilon_{B_i}) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})\Big) \tag{5.8}$$

The specification $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$ can be computed as the most specific one that generalizes all of $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})$, that is the greatest lower bound or *meet* of the lattice formed by them under the refinement ($\sqsupseteq$) relation, whose value was established by lemma 5.

$$(P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) = \sqcap_{i=1...n}(P_{B_i}, Q_{B_i}, \varepsilon_{B_i}) = (\wedge_{i=1...n}P_i, \vee_{i=1...n}Q_i, \cup_{i=1...n}\varepsilon_i) \tag{5.9}$$

Handler abstraction requires that the body (h_i) of each handler refines the handlers specification. For reasoning about **`invoke`** statements inside the body of a handler the non-deterministic choice between the handlers specification and the announced-code specification is used. So it is actually the body of the handler proceed-composed with that non-deterministic choice which should refine the handlers specification.

$$\forall_{i=1...m} \cdot [h_i \circledcirc ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \tag{5.10}$$

There are various ways to establish the handlers' specification leading to different scenarios.

5.4.2.1 II/EA PtolemyRely Modular Scenario

In PtolemyRely language[55], which is an extension of Ptolemy [54, 7], the handler's specification is independent of the base-code specification. Event declarations includes both, handlers specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and announced-code specification, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$.

Assuming *trigger abstraction* (5.8) and *handler abstraction* (5.10), reasoning rules can be formulated. **Announce** statements are reasoned about as the non-deterministic choice between the handlers specification and the current announced-code specification.

(II/EA-PTOLEMYRELY-ANNOUNCE$_R$)

$$\frac{\begin{array}{c}(\textbf{event}\ e\ \{t_1\ var_1,...,t_n\ var_n\ \textbf{relies}\ contract_B\ contract_H\})\in CT,\\ contract_B = \textbf{requires}\ P_{e_B}\ \textbf{modifies}\ \varepsilon_{e_B}\ \textbf{ensures}\ Q_{e_B},\\ contract_H = \textbf{requires}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\ \textbf{assumes}\ A_{e_H}\ \textbf{ensures}\ Q_{e_H},\\ CT,\Gamma\vdash\{P_{B_j}\}\ B_j\ \{Q_{B_j}\}[\varepsilon_{B_j}]\end{array}}{CT,\Gamma\vdash\begin{array}{l}\{P_{e_H}[e_i/var_i]\wedge P_{B_j}\}\\ \quad\textbf{announce}\ e(e_1,\ldots,e_n)\ \{B_j\}\\ \{Q_{e_H}[e_i/var_i]\vee Q_{B_j}\}\\ \left[\varepsilon_{e_B}[e_i/var_i]\cup\varepsilon_{B_j}\right]\end{array}} \tag{5.11}$$

For reasoning about **invoke** statements, the non-deterministic choice between the handlers specification and the base-code specification is used.

(II/EA-PTOLEMYRELY-INVOKE$_R$)

$$\frac{\begin{array}{c}(\textbf{thunk}\ e)=\Gamma(next),\\ (\textbf{event}\ e\ \{t_1\ var_1,...,t_n\ var_n\ \textbf{relies}\ contract_B\ contract_H\})\in CT,\\ contract_B = \textbf{requires}\ P_{e_B}\ \textbf{modifies}\ \varepsilon_{e_B}\ \textbf{ensures}\ Q_{e_B},\\ contract_H = \textbf{requires}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\ \textbf{assumes}\ A_{e_H}\ \textbf{ensures}\ Q_{e_H}\end{array}}{CT,\Gamma\vdash\begin{array}{l}\{(P_{e_H}\wedge P_{e_B})[next.var_i/var_i]\}\\ \quad\textbf{next.invoke}\\ \{(Q_{e_H}\vee Q_{e_B})[next.var_i/var_i]\}\\ \left[(\varepsilon_{e_H}\cup\varepsilon_{e_B})[next.var_i/var_i]\right]\end{array}} \tag{5.12}$$

The PtolemyRely approach allows modular reasoning. The specifications in the event declaration are used to modularly verify handlers and announced-code respectively. It is also flexible, allowing to have separate specifications for handlers and subjects (base code) [55]. However, the base-code reasoning is weakened since the reasoning of **announce** statements is weaker than the reasoning of the corresponding original code, as clearly $(P_{e_H} \wedge P_{B_j},\ Q_{e_H} \vee Q_{B_j},\ \varepsilon_{e_H} \cup \varepsilon_{B_j}) \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$.

5.4.2.2 II/EA Ptolemy Modular Scenario

In Ptolemy language approach the handlers' specification is taken to be the same as the base-code specification:

$$(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) = (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \tag{5.13}$$

As pointed out before (5.11), **announce** statements can be reasoned about as the non-deterministic choice between the handlers' specification and the current announced-code specification. Also, because of *trigger abstraction* (5.8), $B_j \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[e_i/var_i]}$, then from 5.11 it follows that:

$$
\begin{array}{ll}
 & (\textbf{announce}\ e(\ldots)\{B_j\}) \\
\sqsupseteq & \quad \langle \text{by Rule 5.11} \rangle \\
 & (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})_{[e_i/var_i]} \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[e_i/var_i]} \\
= & \quad \langle \text{by } (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) = (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \rangle \\
 & (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[e_i/var_i]} \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[e_i/var_i]} \\
= & \quad \langle \text{by definition of ``}\Box\text{'' and predicate calculus} \rangle \\
 & (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[e_i/var_i]}
\end{array}
$$

yielding the **announce** proof rule.

(II/EA-PTOLEMY-ANNOUNCE$_R$)

$$\frac{(\textbf{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT, \quad contract_H = \ \textbf{requires}\ P_{e_B}\ \textbf{modifies}\ \varepsilon_{e_B}\ \textbf{assumes}\ A_{e_B}\ \textbf{ensures}\ Q_{e_B}}{CT, \Gamma \vdash \{P_{e_B}[e_i/var_i]\}\textbf{announce}\ e(e_1, \ldots, e_n)\ \{B_j\}\{Q_{e_B}[e_i/var_i]\}[\varepsilon_{e_B}[e_i/var_i]]} \tag{5.14}$$

Invoke statements are reasoned about as the non-deterministic choice between the handlers specification and the base-code specification, that in this case can be computed as:

$$\begin{array}{ll} & (\textbf{next.invoke}) \\ \sqsupseteq & \langle \text{by Rule 5.12} \rangle \\ & (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \\ = & \langle \text{by } (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) = (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \text{, definition of ``}\Box\text{'' and predicate calculus} \rangle \\ & (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \end{array}$$

resulting in the **invoke** proof rule.

(II/EA-PTOLEMY-INVOKE$_R$)

$$\frac{(\textbf{thunk}\ e) = \Gamma(next), \ \ (\textbf{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT, \quad contract_H = \ \textbf{requires}\ P_{e_B}\ \textbf{modifies}\ \varepsilon_{e_B}\ \textbf{assumes}\ A_{e_B}\ \textbf{ensures}\ Q_{e_B}}{CT, \Gamma \vdash \begin{array}{c} \{P_{e_B}[next.var_i/var_i]\} \\ \textbf{next.invoke} \\ \{Q_{e_B}[next.var_i/var_i]\} \\ [\varepsilon_{e_B}[next.var_i/var_i]] \end{array}} \tag{5.15}$$

The handlers are reasoned about using handler abstraction (5.10) and using this scenario assumption (5.13). This is calculated as:

$$
\begin{aligned}
& [h_i \circledcirc ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \\
\Leftrightarrow \quad & \langle \text{by } 5.13 \rangle \\
& [h_i \circledcirc ((P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \\
\Leftrightarrow \quad & \langle \text{by definition of ``}\Box\text{'' and predicate calculus} \rangle \\
& [h_i \circledcirc (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})] \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})
\end{aligned}
$$

producing the reasoning obligation (5.16) for the handlers.

$$
\forall_{i=1 \ldots m} \cdot [h_i \circledcirc (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})] \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \tag{5.16}
$$

In summary, the II/EA Ptolemy scenario unifies handler abstraction and trigger abstraction imposing the same specification on handlers and base-code. It allows modular reasoning by using the specification in the event declaration to modularly verify handlers and announced-code. The base-code reasoning is weakened as the reasoning of **`announce`** statements, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, is weaker than the reasoning of the corresponding original code, $(P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$, according to the trigger abstraction adoption, $(P_{B_j}, Q_{B_j}, \varepsilon_{B_j}) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$. That also makes this reasoning approach incomplete, as there exist valid programs that cannot be verified.

5.4.2.3 II/EA Behavior-Preserving Modular Scenario

As illustrated before (Figure 5.5), in single-delivery events the idea is that certain blocks of code of a program are converted into event announcements, passing the block of code to the handlers of the event for its eventual execution. The present scenario is configured with the objective that the behavior of the original program be preserved after event announcement has been added to it. For every block B_j that has been reasoned about against its specification, $(P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$, the corresponding **announce** statement should satisfy that specification. From that, the following computation can be made.

$$\begin{aligned}
& \forall j \cdot (announce\ e(\ldots)\{B_j\}) \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_i/var_i]} \\
\Leftrightarrow \quad & \langle \text{by } B_j \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_i/var_i]} \text{ and rule 5.11} \rangle \\
& \forall j \cdot [(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})_{[e_i/var_i]} \square (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_i/var_i]}] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_i/var_i]} \\
\Leftrightarrow \quad & \langle \text{by equivalence under substitution} \rangle \\
& \forall j \cdot [(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \square (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j}) \\
\Leftrightarrow \quad & \langle \text{by Lemma 8} \rangle \\
& \forall j \cdot (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})
\end{aligned}$$

That is, the specification for the handlers must refine the specification for every announced-block.

$$\forall j \cdot (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j}) \tag{5.17}$$

Then, it can be computed as their least upper bound or *join*, using lemma 6.

$$(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) = \sqcup_{i=1\ldots n}(P_{B_i}, Q_{B_i}, \varepsilon_{B_i}) = (\vee_{i=1\ldots n} P_{B_i}, \wedge_{i=1\ldots n}(old(P_{B_i}) \Rightarrow Q_{B_i}), \cap_{i=1\ldots n} \varepsilon_{B_i}) \tag{5.18}$$

By *trigger abstraction* (5.8) $(P_{B_j}, Q_{B_j}, \varepsilon_{B_j}) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, then from 5.17 it follows that

$$(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \tag{5.19}$$

Assuming *trigger abstraction* (5.8), *handler abstraction* (5.10) and 5.17, the reasoning rules can be formulated. The **announce** rule is as follows 5.20. By construction it guarantees behavior preservation.

$$\frac{\begin{array}{l}\text{(II/EA-BehaviorPreserving-Announce}_R\text{)}\\ (\texttt{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\ \texttt{relies}\ contract_B\ contract_H\}) \in CT,\\ \qquad\qquad \{P_{B_j}\}B_j\{Q_{B_j}\}[\varepsilon_{B_j}]\end{array}}{CT, \Gamma \vdash \{P_{B_j}\}\texttt{announce}\ e(e_1, \ldots, e_n)\ \{B_j\}\{Q_{B_j}\}[\varepsilon_{B_j}]} \tag{5.20}$$

From 5.19, lemma 8 implies that $[(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})] \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$. Using this and the general **invoke** rule (5.12), it follows that
$(\texttt{next.invoke}) \sqsupseteq ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$,
as expressed by the following **invoke** rule (5.21).

$$\frac{\begin{array}{c}\text{(II/EA-BehaviorPreserving-Invoke}_R\text{)}\\ (\texttt{thunk}\ e) = \Gamma(next),\\ (\texttt{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\ \texttt{relies}\ contract_B\ contract_H\}) \in CT,\\ contract_B = \texttt{requires}\ P_{e_B}\ \texttt{modifies}\ \varepsilon_{e_B}\ \texttt{ensures}\ Q_{e_B},\\ contract_H = \texttt{requires}\ P_{e_H}\ \texttt{modifies}\ \varepsilon_{e_H}\ \texttt{assumes}\ A_{e_H}\ \texttt{ensures}\ Q_{e_H}\end{array}}{\begin{array}{c}\{P_{e_B}[next.var_i/var_i]\}\\ CT, \Gamma \vdash \quad \texttt{next.invoke}\\ \{Q_{e_B}[next.var_i/var_i]\}\\ [\varepsilon_{e_B}[next.var_i/var_i]]\end{array}} \tag{5.21}$$

According to 5.10 handlers are reasoned about as $[h_i \circledcirc ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. In this case that corresponds to $[h_i \circledcirc (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$.

This approach is also modular. It eases verification of subjects (original code). It is demanding for the handlers, as their behavior must refine the behavior of all the blocks of announced code.

5.4.2.4 II/EA Non-Modular Individual Refinement Scenario

Non-modular reasoning requires a case by case analysis, instead of using abstraction. It is more precise but requires whole-program analysis, and so it is more time consuming. In the case of II/EA Single Delivery, for reasoning about an event announcement it is required to consider, case by case, each one of the handlers for the event. One extra consideration that must be taken into account is that each handler depends on other handlers and on the announced code, because of **`invoke`** statements in its body.

This scenario explores a compromise in which, at each event announcement of the form **`announce`** $e(\ldots)\ \{B_j\}$, with $B_j \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_i/var_i]}$, every handler h_i is reasoned about *in isolation* from other handlers, only considering the behavior of the announced code. The handler is reasoned about against the announced-code specification, so it preserves that behavior.

$$[\, b_{i[var_t/next_i.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\,] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j}), \quad \text{where } h_i \equiv \big(m_i(t_e\ next_i)\{b_i\}\big) \tag{5.22}$$

The following lemma extends this individual refinement property to a chain of handlers.

Lemma 10 (Handler chain refinement)**.**

$$\left[\begin{array}{c} \Big(\forall i_{=1..m} : h_i \equiv \big(m_i(t_e\ next_i)\{b_i\}\big) \bullet \\ [\ b_{i[var_t/next_i.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\ ] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\Big) \\ \Rightarrow \Big([b_{1[var_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[var_t/next_k.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\Big) \\ \textit{for any permutation } (b_1, b_2, \ldots, b_k) \textit{ of } k \textit{ out of } m \textit{ handlers} \end{array}\right] \begin{array}{c} (10.1) \\ (10.2) \end{array}$$

Proof: (By induction on the number of the handlers: k)

Case 1. $[k = 1]$

$b_{1[var_t/next_1.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$

$\sqsupseteq$ ⟨by hypothesis, with $[h_1]$ a permutation of 1 out of the m handlers⟩

$(P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$ ∎

Case 2. [assume for k and prove for $k+1$. Let $(h_1, h_2, \ldots, h_k, h_{k+1})$ be a permutation of $k+1$ out of m handlers.]

$b_{1[var_t/next_1.var_t]} \circledcirc b_{2[var_t/next_2.var_t]} \circledcirc \cdots \circledcirc b_{k+1[var_t/next_{k+1}.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$

$=$ ⟨by definition of composition $\circledcirc$⟩

$b_{1[var_t/next_1.var_t]} \circledcirc [b_{2[var_t/next_2.var_t]} \circledcirc \cdots \circledcirc b_{k+1[var_t/next_{k+1}.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})]$

$\sqsupseteq$ ⟨by the induction hypothesis over $(h_2 \ldots h_{k+1})$ and monotonicity of refinement ($\sqsupseteq$) [3, 45]⟩

$b_{1[var_t/next_1.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$

$\sqsupseteq$ ⟨by hypothesis 10.1 ⟩

$(P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$ ∎

Lemma 11 (Handler refinement in base code)**.**

$$\left[\begin{array}{c} \Big(\forall i_{=1..m} : h_i \equiv \big(m_i(t_e\; next_i)\{b_i\}\big) \bullet \\ [\; b_{i[var_t/next_i.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\;] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\Big) \\ \Rightarrow \Big([b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[e_t/next_k.var_t]} \circledcirc B_j] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}\Big) \\ \textit{for any permutation } (b_1, b_2, \ldots, b_k) \textit{ of } k \textit{ out of } m \textit{ handlers} \end{array}\right] \quad \begin{array}{c} (11.1) \\ (11.2) \end{array}$$

Proof: (Handler refinement in base code)

$$\begin{array}{ll} & b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[e_t/next_k.var_t]} \circledcirc B_j \\ \sqsupseteq & \langle \text{by } B_j \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}] \rangle \\ & b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[e_t/next_k.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]} \\ = & \langle \text{by application of mutually cancelling substitutions } [var_t/e_t] \text{ and } [e_t/var_t] \rangle \\ & \Big(b_{1[var_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[var_t/next_k.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\Big)[e_t/var_t] \\ \sqsupseteq & \langle \text{by lemma 10} \rangle \\ & (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})[e_t/var_t] \end{array}$$

According to the semantics of **`announce`**, the following holds:

$$(announce\; e(e_1, .., e_n)\; \{B_j\}) \equiv \begin{cases} B_j, \text{there are no handlers} \\ [b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k[e_t/next_k.var_t]} \circledcirc B_j], \text{there are handlers} \end{cases}$$

It can be shown that $(announce\; e(e_1, .., e_n)\; \{B_j\}) \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}$. In the first case it follows from $B_j \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}$. In the second case it follows from lemma 11.

This is expressed in the following reasoning rule for **announce** statement, where the function $handlersOf()$ returns the set of handlers for an event.

(II/EA-INDIVIDUALREFINEMENT-ANNOUNCE$_R$)

$$\frac{\begin{array}{c}(\textbf{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT, \\ CT, \Gamma \vdash \{P_{B_j[e_t/var_t]}\}\ B_j\ \{Q_{B_j[e_t/var_t]}\}[\varepsilon_{B_j[e_t/var_t]}], \\ \forall h_i \in handlersOf(e) : h_i \equiv \big(m_i(t_e\ next_i)\{b_i\}\big) \bullet \\ CT, \Gamma \vdash [b_{i[var_t/next_i.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\end{array}}{CT, \Gamma \vdash \{P_{B_j[e_t/var_t]}\}\ \textbf{announce}\ e(e_1, \ldots, e_n)\ \{B_j\}\ \{Q_{B_j[e_t/var_t]}\}[\varepsilon_{B_j[e_t/var_t]}]} \tag{5.23}$$

An **invoke** statement is always executed as part of the body of a handler, in the execution chain for an announcement of the form $announce\ e(e_1, .., e_n)\ \{B_j\}$ for an event e. The semantics of **invoke**, is such that the following holds:

$$(next.invoke)_{[e_t/next.var_t]} \equiv \begin{cases} B_j, \text{there are no more handlers.} \\ [b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k'[e_t/next_{k'}.var_t]} \circledcirc B_j], \text{pending handlers.} \\ \quad (h_1, \ldots, h_{k'}) \text{ is a permutation of } k' < k \text{ of the } k \text{ handlers for } e. \end{cases}$$

From that, it follows that $(next.invoke)_{[var_t/next.var_t]} \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$. In the first case,

$$\begin{array}{l} \quad (next.invoke)_{[e_t/next.var_t]} \equiv B_j \\ \Longrightarrow\ \langle \text{by } B_j \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}] \rangle \\ \quad (next.invoke)_{[e_t/next.var_t]} \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}] \\ \Longrightarrow\ \langle \text{by applying substitution } [var_t/e_t] \text{ in both sides} \rangle \\ \quad \Big((next.invoke)_{[e_t/next.var_t]}\Big)_{[var_t/e_t]} \sqsupseteq \Big((P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}]\Big)_{[var_t/e_t]} \\ \Longrightarrow\ \langle \text{by composing multiple substitutions} \rangle \\ \quad (next.invoke)_{[var_t/next.var_t]} \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j}) \end{array}$$

In the second case,

$$(next.invoke)_{[e_t/next.var_t]} \equiv [b_{1[e_t/next_1.var_t]} \circledcirc \cdots \circledcirc b_{k'[e_t/next_{k'}.var_t]} \circledcirc B_j]$$

$\Longrightarrow$ ⟨by lemma 11⟩

$$(next.invoke)_{[e_t/next.var_t]} \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}$$

$\Longrightarrow$ ⟨by applying substitution $[var_t/e_t]$ in both sides⟩

$$\Big((next.invoke)_{[e_t/next.var_t]}\Big)_{[var_t/e_t]} \sqsupseteq \Big((P_{B_j}, Q_{B_j}, \varepsilon_{B_j})_{[e_t/var_t]}\Big)_{[var_t/e_t]}$$

$\Longrightarrow$ ⟨by composing multiple substitutions⟩

$$(next.invoke)_{[var_t/next.var_t]} \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})$$

This is expressed in the following reasoning rule for **invoke** statement.

$$\text{(II/EA-IndividualRefinement-Invoke}_R\text{)}$$

$$\frac{\begin{array}{c}\Gamma(next) = e,\ \ (\textbf{event}\ e\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT, \\ announcedBlock(next) = B_j, \\ announcedExpressions(next) = (e_1, .., e_n), \\ \{P_{B_j[e_t/var_t]}\}\ B_j\ \{Q_{B_j[e_t/var_t]}\}[\varepsilon_{B_j[e_t/var_t]}], \\ \forall h_i \in handlersOf(e) : h_i \equiv \big(m_i(t_e\ next_i)\{b_i\}\big) \bullet \\ CT, \Gamma \vdash [b_{i[var_t/next_i.var_t]} \circledcirc (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})] \sqsupseteq (P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\end{array}}{CT, \Gamma \vdash \{P_{B_j}\}\ \textbf{next.invoke}_{[var_t/next.var_t]}\ \{Q_{B_j}\}[\varepsilon_{B_j}]} \tag{5.24}$$

The functions $announcedBlock(next)$ and $announcedExpressions(next)$ returns the information in the closure for the most recent announcement of for event e: the announced code and the announced expressions respectively.

This scenario is not modular because at each announcement all the handlers for the corresponding event must be reasoned about. However it keeps the original behavior and only requires that each handler be reasoned about in isolation, without considering all the possible orders of execution.

5.5 AO Scenarios

In AO scenarios the blocks $\{B_i\}$ in the reference scenario (Fig. 5.1) correspond to shadows of join points, that is, actual pieces of code considered to trigger the join point events [29]. In Figure 5.8 the highlighted lines on the left (lines 7,9 and 11) represent shadows of pointcuts defined on the right (lines 16 and 22).

```
1  class C{ ...
2   //@ requires Ps;
3   //@ modifies εs;
4   //@ ensures Qs;
5   s(){
6     ...
7     {B1}
8     ...
9     {Bj}
10    ...
11    {Bn}
12    ...
13  } ...
14 }
```

```
15 aspect A{
16  pointcut pca() : ...; //pc expression
17  around pca(){ // advice
18   ...
19   proceed();
20   ...
21  }
22  pointcut pcb() : ...; //pc expression
23  around pcb(){ // advice
24   ...
25   proceed();
26   ...
27  }
28 }
```

Figure 5.8: AO Scenario

Figure 5.9 presents a concrete AO example. A shadow could be an assignment statement (lines 7 and 9), matching a *field-set* join point (line 44), a method invocation statement (line 8), corresponding to a *method-call* join point (line 29) or even a complete method body (line 18), corresponding to a *method-execution* join point (line 36).

```
 1 class BaseCode {
 2  int a;
 3  PrintStream out=
 4    System.out;
 5
 6  public void run(){
 7   a=1;
 8   methodA(a);
 9   a=2;
10   methodB(a);
11  }
12
13  void methodA(int f){
14   out.println("Method A");
15  }
16
17  void methodB(int f){
18   out.println("Method B");
19  }
20
21  static void main(
22     String[] args){
23   new BaseCode().run();
24  }
25 }
```

```
26 aspect AnAspect {
27  PrintStream out=System.out;
28  pointcut callPointcut(int p):
29   call(void *.methodA(int))&&args(p);
30  void around(int p):callPointcut(p){
31   out.println("Start Call: "+p);
32   proceed(p);
33   out.println("End Call");
34  }
35  pointcut execPointcut(int p):
36   execution(void *.methodB(int))
37     &&args(p);
38  void around(int p):execPointcut(p){
39   out.println("Start Exec.");
40   proceed(p);
41   out.println("End Exec.");
42  }
43  pointcut setPointcut(int p):
44   set(int BaseCode.a)&&args(p);
45  void around(int p):setPointcut(p){
46   out.println("Start Set: "+p);
47   proceed(p);
48   out.println("End Set");
49  }
50 }
```

Figure 5.9: AO Shadows

In **around** advice, pieces of code from aspects (lines 30-34, 38-42 and 45-49) are executed *instead of* the blocks of base code shadowed by the corresponding pointcuts. In the body of a piece of advice, a *proceed* instruction (lines 32, 40 and 47) executes the original shadowed block, or other pieces of advice shadowing the same block. Once the shadows of the ponintcuts have been identified, the pieces of advice are woven into the corresponding places. At runtime, the chain of pieces of advice matching a shadowed block of code is executed, according to the *proceed* instructions in their bodies. This is illustrated in Figure 5.10.

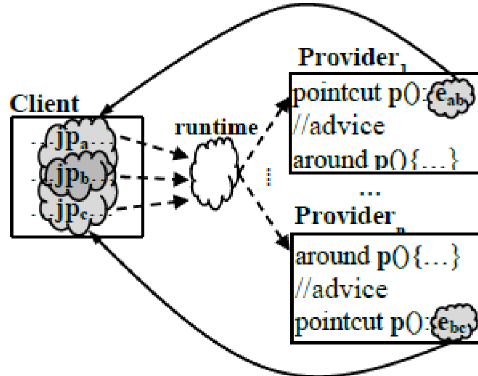

Figure 5.10: AO at Runtime

This AO scenario is similar to the previous II scenarios. The main difference is that here the events are considered to be triggered not explicitly by **`announce`** statements but implicitly by the execution of the shadows picked by pointcuts. An analogy can be made between AO and II scenarios [62]. The pointcut declaration in AO corresponds to the event declaration in II. The shadows in AO corresponds to the blocks of announced code in II. The pieces of advice correspond to the handlers in II. *Proceed* instructions in AO correspond to *invoke* instructions in II.

Due to the similarities between AO and II, reasoning about AO scenarios can be tackled by using II strategies, following a two-phase approach. First, for each join point in the base code all matching pointcuts are identified, and all the pieces of advice for those pointcuts are considered the handlers for the join point event. Then the reasoning strategies used for the previous *single delivery II/EA* scenarios can be used. For doing modular reasoning, specification features like the ones in PtolemyRely can be applied. A pointcut definition, e, can be annotated with the specification, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, that matching join point shadows must satisfy and with the specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, that every implementing piece of advice should refine. The specifications for the shadowed blocks of code depend on the type of join points. For *method-call* and *method-execution* join points the specification of the corresponding method can be used, $(P_m, Q_m, \varepsilon_m)$. For *field-set* join points ($x = e$) the usual assignment specification $(P[e/x], P, \{x\})$ can be used; and so on.

CHAPTER 6: EVALUATION

The evaluation of the results from this work, as a solution to the proposed research problem, is performed by various means.

The theorems about the soundness of the proof rules that apply in each scenario constitute a first evaluation mechanism. Also case studies involving the reasoning scenarios are presented in order to assess the usefulness of the corresponding proof rules. These case studies correspond to small programs that are manually reasoned about using the proof rules that apply in each case. The programs were selected considering their appropriateness to show the pros and cons of each configuration of tradeoffs.

6.1 Soundness of Scenarios' Proof Rules

In this section the soundness of the essential proof rules for the different scenarios is demonstrated. The soundness for standard object oriented rules is assumed and can be found in various sources [1, 18, 50, 39].

6.1.1 Single Delivery II: PtolemyRely

A small-step operational semantics for PtolemyRely [55] is presented. Using this semantics the soundness of the corresponding proof rules is then demonstrated. This semantics is based on the original semantics of Ptolemy [53] and some adaptations of it [6, 4]. The execution state is represented by a configuration $\langle \mathbb{C}[c], S, \Pi, A \rangle$, containing the program ($\mathbb{C}[c]$) and a command (c) to execute, a store representing the current memory state (S), a store typing (Π) and the list of

active handlers (A). The *semantics* is expressed as a *evaluation* relation between configurations, meaning that executing one step of statement causes a transition from one configuration to another. We distinguish between methods, m, and procedures, p. Methods return a value and are side-effect free, and then can be used in side-effect free expressions. Procedures do not return a value and can have side effects, and then can be used as commands but not in expressions.

Evaluation contexts for expressions ($\mathbb{E}$) and commands ($\mathbb{C}$) :

$$
\begin{array}{ll}
\mathbb{E} ::= & - \mid \mathbb{E}.m(e\ldots) \mid v.m(v\ldots\mathbb{E}\,e\ldots) \mid \mathbb{E}.f \mid \textbf{cast}\ t\ (\mathbb{E}) \\
\mathbb{C} ::= & - \mid \mathbb{E}.f = e \mid v.f = \mathbb{E} \mid x = \mathbb{E} \mid \mathbb{E}.p(e\ldots) \mid v.p(v\ldots\mathbb{E}\,e\ldots) \\
& \mid \textbf{if}(\mathbb{E})\ \{c\}\ else\ \{c\} \mid \textbf{while}(\mathbb{E})\ \{c\} \mid t\ var = \mathbb{E}; c \mid \mathbb{C}; c \\
& \mid \textbf{announce}\ ev(v\ldots\mathbb{E}\,e\ldots)\{c\} \mid \textbf{invoke}(\mathbb{E}) \\
& \mid \textbf{register}(\mathbb{E}) \mid \textbf{unregister}(\mathbb{E})
\end{array}
$$

Evaluation relation: $\hookrightarrow$: $\langle \mathbb{C}[c], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[c'], S', \Pi', A'\rangle$

Domains:

$\Sigma ::= \langle \mathbb{C}[c], S, \Pi, A\rangle$	“Configurations”
$S ::= \{loc_k \mapsto sv_k\} k \in K$	“Stores”
$v ::= null \mid loc$	“Values”
$sv ::= or \mid ec$	“Storable Values”
$or ::= [C.F]$	“Object Records”
$F ::= \{f_k \mapsto v_k\} k \in K$	“Field Maps”
$\rho ::= \{var_k \mapsto v_k\} k \in K$	“Environments”
$ec ::= \textbf{eClosure}(H, c, \rho)$	“Event Closures”
$H ::= H + h \mid \bullet$	“Handler Record List”
$h ::= \langle loc, m\rangle$	“Handler Record”
$A ::= \{ev_k \mapsto O_k\} k \in K$	“Active Objects Map”
$O ::= loc + O \mid \bullet$	“Active Objects List”

where K is finite.

Figure 6.1: PtolemyRely’s evaluation contexts and configuration.

A value v is either a location loc or $null$. The store maps locations to storable values sv, which are either objects records or or event closures ec. An object record has a class name C and a map F from fields to values. An event closure $\textbf{eClosure}(H, c, \rho)$ contains an ordered list of handlers H, a command c, that corresponds to a block of announced-code, and an environment ρ. A handler record h contains a location loc, which points to an observer object, and a method name m.

The dynamic semantic rules of PtolemyRely are shown in Figure 6.2. The class table CT is the set of all class and event type declarations in the program.

$$\textsc{(PtolemyRely-Announce}_S\textsc{)}\quad \frac{\begin{array}{c}(\texttt{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \texttt{relies}\ contract_B\ contract_H\}) \in CT, \\ H = handlersOf(ev, A),\ \rho = \{var_i \mapsto v_i\}_{i=1}^{n},\ loc \notin dom(S), \\ S' = S \uplus (loc \mapsto \texttt{eClosure}(H, c, \rho)), \\ \Pi' = \Pi \uplus \{loc\ :\ \texttt{var}\ \texttt{thunk}\ ev\}, \\ A' = A\end{array}}{CT \vdash \langle \mathbb{C}[\texttt{announce}\ ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[\texttt{invoke}\ loc], S', \Pi', A'\rangle}$$

$$\textsc{(PtolemyRely-InvokeDone}_S\textsc{)}\quad \frac{\texttt{eClosure}(\bullet, c, \rho) = S(loc)}{CT \vdash \langle \mathbb{C}[\texttt{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[c], S, \Pi, A\rangle}$$

$$\textsc{(PtolemyRely-Invoke}_S\textsc{)}\quad \frac{\begin{array}{c}(\texttt{thunk}\ ev) = \Pi(loc), \\ (\texttt{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \texttt{relies}\ contract_B\ contract_H\}) \in CT, \\ \texttt{eClosure}(H, c, \rho) = S(loc),\ \ H = \langle loc_h, m_h\rangle + H', \\ [C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h),\ \ loc' \notin dom(S), \\ S' = S \uplus (loc' \mapsto \texttt{eClosure}(H', c, \rho)), \\ \Pi' = \Pi \uplus \{loc'\ :\ \texttt{var}\ \texttt{thunk}\ ev\}, \\ A' = A\end{array}}{CT \vdash \langle \mathbb{C}[\texttt{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^{n},\ ^{loc'}/_{var_{h_1}},\ ^{loc_h}/_{this}]\big], S', \Pi', A'\rangle}$$

Figure 6.2: PtolemyRely's Semantics.

The ($\textsc{PtolemyRely-Announce}_S$) rule creates a closure that includes the list of handlers for the event, the command that represents the block of announced code and an environment mapping the context variables to its actual values. Then it stores that closure in a fresh location loc and starts running this closure by invoking that location. The ($\textsc{PtolemyRely-InvokeDone}_S$) rule applies when there are no more handlers in the closure, and so it executes the command corresponding to the originally announced block of code. The ($\textsc{PtolemyRely-Invoke}_S$) rule executes the body c_h of the first handler in the closure. First, it puts the rest of the closure (remaining handlers and same environment and announced-code command) into a fresh location loc', and substitutes in that

body c_h any reference var_{h_1} to the original closure by a reference to the new closure loc'. In this way any **invoke** statement in the changed body c'_h will execute the next handler, which will be the first one in the new closure. Every reference to a context field $var_{h_1}.var_i$ is substituted by the corresponding field in the environment $\rho(var_i)$. It also substitutes any reference to **this** by a reference to the observer object loc_h containing the handler method.

Before demonstrating the soundness of the proof rules 5.11 and 5.12 for PtolemyRely, some definitions, axioms and theorems are presented. In the PotlemyRely proof system programs are regarded as acting upon variables, and so a typing context Γ is used. Furthermore, in the semantics of PtolemyRely the programs are regarded as manipulating locations on a store, instead of variables, and so a store typing Π is used.

Definition 12. *(Valid States)*

A state (S, Π, A), where S is a store, Π a typing environment and A a list of handlers, is valid, denoted as $valid(S, \Pi, A)$, iff:

$$\Big((dom(S) = dom(\Pi)) \wedge (\forall\, loc \in dom(S) \bullet \Pi(loc) = typeof(S(loc)))\Big)$$
$$\wedge \Big(\forall\, loc \in dom(S), ev \in CT \bullet \Big(\big((\Pi(loc) = (\textbf{thunk}\ ev)) \wedge (S(loc) = \textbf{eClosure}(H, c, \rho))\big) \Rightarrow \big(\forall\, \langle loc_h, m_h \rangle \in H \bullet loc_h \in A\big)\Big)\Big)$$

Definition 13. *(Changed Locations)*

The set of changed locations from store S to store S' is:

$$\Delta(S, S') = \{l \in dom(S) \bullet S(l) \neq S'(l)\}$$

Definition 14. *(Semantic Refinement)*

A statement, c, refines specification, (P, Q, ε), denoted as $c \mathrel{\mathring{\sqsupseteq}} (P, Q, \varepsilon)$, iff

$$\left[\begin{array}{c} \forall\, S, \Pi, A : valid(S, \Pi, A) \bullet \\ \langle \mathbb{C}[c], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle \Longrightarrow \big((\llbracket P \rrbracket_S \Rightarrow \llbracket Q \rrbracket_{S,S'}) \wedge (\Delta(S, S') \subseteq \llbracket \varepsilon \rrbracket_S)\big) \end{array}\right]$$

Reasoning about handlers is more involved. The body of a handler method m_h, that belongs to a handler object located at a given location loc_h, may have **invoke** statements on it. Every one of them will invoke the same closure $\textbf{eClosure}(H, c, \rho)$ stored at certain location loc of the store S. As the execution of the body of the handler proceeds it will reach each one of the invocations at a different configuration $\langle \mathbb{C}[\textbf{invoke}\ loc], S_k, \Pi_k, A_k\rangle$. If the execution of each one of these invocations satisfies a given specification (P, Q, ε), then, for reasoning purposes, this specification can substitute the actual invocations in the handler, that is called their *proceed-composition* $\langle loc_h, m_h\rangle \circledcirc (P, Q, \varepsilon)$. The handler method, with this substitution, can be reasoned about using its expected specification (P', Q', ε'). This is formalized in the following definition.

Definition 15. *(Semantic Refinement Of Proceed-Composition)*
A handler $\langle loc_h, m_h\rangle$ *proceed-composed with a specification* (P, Q, ε) *refines another specification* (P', Q', ε')*, denoted as* $\langle loc_h, m_h\rangle \circledcirc (P, Q, \varepsilon) \mathrel{\mathring{\sqsupseteq}} (P', Q', \varepsilon')$*, iff* $\forall\, S, \Pi, A : valid(S, \Pi, A)\ \bullet$

$$\left[\begin{array}{c}
\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\
\Big([C_h.F_h] = S(loc_h)\Big) \ \wedge\ \Big((t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\
\Big((t_1\ var_1, ..., t_n\ var_n) = varsOf(ev, CT)\Big) \wedge \\
\Big((\textbf{thunk}\ ev) = \Pi(loc), \textbf{eClosure}(H, c, \rho) = S(loc)\Big) \wedge \\
\left(\begin{array}{c}
\forall S_k, \Pi_k, A_k, \mathbb{C}_k\ s.t.\ S_k(loc) = S(loc), \Pi_k(loc) = \Pi(loc), A_k = A\ \bullet \\
\Bigg(\Big(\langle \mathbb{C}_k[\textbf{invoke}\ loc], S_k, \Pi_k, A_k\rangle \hookrightarrow^* \langle \mathbb{C}_k[skip], S_k', \Pi_k', A_k'\rangle\Big) \\
\Longrightarrow \Big((\llbracket P_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S_k} \Rightarrow \llbracket Q_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S_k, S_k'}) \\
\wedge\, (\Delta(S_k, S_k') \subseteq \llbracket \varepsilon_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S_k})\Big)\Bigg)
\end{array}\right)
\end{array}\right]$$

$$\Longrightarrow \left[\begin{array}{c}
\Big(\langle \mathbb{C}\big[c_h[(\tfrac{\rho(var_i)/}{var_{h_1}.var_i})_{i=1}^n,\ \tfrac{loc'/}{var_{h_1}},\ \tfrac{loc_h/}{this}]\big], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \\
\Longrightarrow \Big((\llbracket P'_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket Q'_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}) \wedge (\Delta(S, S') \subseteq \llbracket \varepsilon'_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S)\Big)
\end{array}\right]$$

The function $varsOf(ev, CT)$ is defined as follows:

$$\begin{gathered}(t_1\ var_1, ..., t_n\ var_n) = \mathbf{\texttt{varsOf}}(ev, CT) \equiv \\ (\mathbf{\texttt{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \mathbf{\texttt{relies}}\ contract_B\ contract_H\}) \in CT\end{gathered} \tag{6.1}$$

We assume *trigger abstraction* (5.8), where every piece of announced code e satisfies the base code specification, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, given in the event declaration, ev. That is:

Axiom 16 (Announced code satisfies specification)**.**

$$\begin{array}{l}\forall\ S, \Pi, A : valid(S, \Pi, A) \bullet \\ \left[\begin{array}{c}\Big((\mathbf{\texttt{thunk}}\ ev) = \Pi(loc)\Big) \wedge \\ \Big((\mathbf{\texttt{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \mathbf{\texttt{relies}}\ contract_B\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \mathbf{\texttt{requires}}\ P_{e_B}\ \mathbf{\texttt{modifies}}\ \varepsilon_{e_B}\ \mathbf{\texttt{ensures}}\ Q_{e_B}\Big) \wedge \\ \Big(contract_H = \mathbf{\texttt{requires}}\ P_{e_H}\ \mathbf{\texttt{modifies}}\ \varepsilon_{e_H}\ \mathbf{\texttt{assumes}}\ A_{e_H}\ \mathbf{\texttt{ensures}}\ Q_{e_H}\Big) \wedge \\ \Big(\mathbf{\texttt{eClosure}}(H, c, \rho) = S(loc)\Big)\end{array}\right] \\ \Longrightarrow \left[c \mathrel{\mathring{\sqsupseteq}} (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})_{[\rho(var_i)/var_i]_{i=1}^{n}}\right]\end{array}$$

We also assume *handler abstraction* (5.10), where every handler body proceed-composed with the non-deterministic choice of the handlers and announced-code specifications, satisfies the handlers specification: $[e_h \circledcirc ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. That is, if the body e_h of a handler method m_h is executed upon a handler object(at loc_h) and every **`invoke`** statement on it, when executed upon an event closure (at loc), satisfies the non-deterministic choice between the handlers and base code specifications for the corresponding event then that handler body e_h satisfies the handlers specification for the event.

Axiom 17 (Handlers satisfy specification)**.**

$$\forall\, S, \Pi, A : valid(S, \Pi, A) \bullet$$
$$\left[\begin{array}{c} \Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big((\textbf{event}\; ev\; \{t_1\; var_1, ..., t_n\; var_n\; \textbf{relies}\; contract_B\; contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \textbf{requires}\; P_{e_B}\; \textbf{modifies}\; \varepsilon_{e_B}\; \textbf{ensures}\; Q_{e_B}\Big) \wedge \\ \Big(contract_H = \;\textbf{requires}\; P_{e_H}\; \textbf{modifies}\; \varepsilon_{e_H}\; \textbf{assumes}\; A_{e_H}\; \textbf{ensures}\; Q_{e_H}\Big) \end{array}\right]$$

$$\Longrightarrow \Big[[\langle loc_h, m_h\rangle \circledcirc ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H})\Box(P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \mathrel{\mathring{\sqsupseteq}} (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})\Big]$$

We use axioms 16 and 17 to prove the following theorem.

Theorem 18 (Invoke Reasoning)**.** *For all valid states* (S, Π, A) *the following holds: given an event,* ev*, and an closure for it, stored at a location* loc*,* $\textbf{eClosure}(H, c, \rho) = S(loc)$*, such that the expression* c *satisfies the specification* $(P_c, Q_c, \varepsilon_c)$ *(besides satisfying the event's base-code specification), then an* **invoke** *statement of the form* **invoke** loc *refines the non-deterministic choice of the handlers and base-code specifications for* ev *and also refines the non-deterministic choice of the handlers specification for* ev *and the given specification* $(P_c, Q_c, \varepsilon_c)$*. That is:*

$$\left[\begin{array}{c} \Big((\textbf{thunk}\; ev) = \Pi(loc)\Big) \wedge \\ \Big((\textbf{event}\; ev\; \{t_1\; var_1, ..., t_n\; var_n\; \textbf{relies}\; contract_B\; contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \textbf{requires}\; P_{e_B}\; \textbf{modifies}\; \varepsilon_{e_B}\; \textbf{ensures}\; Q_{e_B}\Big) \wedge \\ \Big(contract_H = \;\textbf{requires}\; P_{e_H}\; \textbf{modifies}\; \varepsilon_{e_H}\; \textbf{assumes}\; A_{e_H}\; \textbf{ensures}\; Q_{e_H}\Big) \wedge \\ \Big(\textbf{eClosure}(H, c, \rho) = S(loc)\Big) \wedge \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \end{array}\right] \begin{array}{r} (18.1) \\ (18.2) \\ (18.3) \\ (18.4) \\ (18.5) \\ (18.6) \end{array}$$

$\Longrightarrow$

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\textbf{\textit{invoke}}\, loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big) \\ \Rightarrow \left(\begin{array}{c}\Big(\llbracket (P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket (Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \cup \llbracket \varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S)\Big) \wedge \\ \Big(\llbracket (P_{e_H[\rho(var_i)/var_i]_{i=1}^n} \wedge P_c)\rrbracket_S \Rightarrow \llbracket (Q_{e_H[\rho(var_i)/var_i]_{i=1}^n} \vee Q_c)\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \cup \llbracket \varepsilon_c\rrbracket_S)\Big)\end{array}\right)\end{array}\right] \tag{18.7}$$

Proof: [by induction over the length of the list of handlers $|H|$]

Case 1. $[|H| = 0 \ \ i.e.\ H = \bullet]$

$\langle$by 18.7$\rangle$

$$\Big[\langle \mathbb{C}[\textbf{invoke}\, loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big]$$

$\langle$by hypothesis 18.1, 18.5 and case hypothesis$\rangle$

$$\left[\begin{array}{c}\Big(\textbf{eClosure}(\bullet, c, \rho) = S(loc)\Big) \wedge \\ \Big(\langle \mathbb{C}[\textbf{invoke}\, loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by (PTOLEMYRELY-INVOKEDONE$_S$) semantic rule and transitivity of $\hookrightarrow$ $\rangle$

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\textbf{invoke}\, loc], S, \mathit{\Pi}, A\rangle \hookrightarrow \langle \mathbb{C}[c], S, \mathit{\Pi}, A\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[c], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by axiom 16: $c \mathrel{\mathring{\sqsupseteq}} (P_{eB}, Q_{eB}, \varepsilon_{eB})$, hypothesis 18.6: $c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)$ and definition 14 $\rangle$

$$\left[\begin{array}{c}\Big(\llbracket P_{e_B[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket Q_{e_B[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S\Big) \wedge \\ \Big(\llbracket P_c\rrbracket_S \Rightarrow \llbracket Q_c\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_c\rrbracket_S\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by predicate calculus and set theory$\rangle$

$$\left[\begin{array}{c}\Big(\llbracket (P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket (Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \cup \llbracket \varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S\Big) \wedge \\ \Big(\llbracket (P_{e_H[\rho(var_i)/var_i]_{i=1}^n} \wedge P_c)\rrbracket_S \Rightarrow \llbracket (Q_{e_H[\rho(var_i)/var_i]_{i=1}^n} \vee Q_c)\rrbracket_{S,S'}\Big) \wedge \\ \Big(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \cup \llbracket \varepsilon_c\rrbracket_S\Big)\end{array}\right] \blacksquare$$

Case 2. [induction: $H = \langle loc_h, m_h\rangle + H''$, assume for H'' and prove for H]

$\langle$by 18.7$\rangle$

$$\Big[\langle \mathbb{C}[\texttt{invoke}\ loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big]$$

$\langle$by case hypothesis and 18.1, 18.5 and **eClosure** well-formedness $\rangle$

$$\left[\begin{array}{c}\Big(H = \langle loc_h, m_h\rangle + H''\Big) \wedge \\ \Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big(\langle \mathbb{E}[\texttt{invoke}\ loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{E}[skip], S', \mathit{\Pi}', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by 18.1, 18.2, 18.5, (PTOLEMYRELY-INVOKE$_S$) semantic rule and transitivity of $\hookrightarrow$ $\rangle$

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big(loc'' \notin dom(S)\Big) \wedge \\ \Big(\Pi'' = \Pi \uplus \{loc''\ :\ var\ \texttt{thunk}\ ev\}\Big) \wedge \\ \Big(S'' = S \uplus (loc'' \mapsto \texttt{eClosure}(H'', c, \rho))\Big) \wedge \\ \Big(A'' = A\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ ^{loc''}/_{var_{h_1}},\ ^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ ^{loc''}/_{var_{h_1}},\ ^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by construction of loc'', S'', Π'' and H''⟩

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\texttt{thunk}\ ev) = \Pi''(loc'')\Big) \ \wedge\ \Big(\texttt{eClosure}(H'', c, \rho) = S''(loc'')\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ ^{loc''}/_{var_{h_1}},\ ^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by the inductive hypothesis applied to every [$\texttt{invoke}\ loc''$] in c_h⟩

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\texttt{thunk}\ ev) = \Pi''(loc'')\Big) \ \wedge\ \Big(\texttt{eClosure}(H'', c, \rho) = S''(loc'')\Big) \wedge \\ \left(\begin{array}{c}\forall\ S_k, \Pi_k, A_k, \mathbb{C}_k\ s.t.\ S_k(loc'') = S''(loc''), \Pi_k(loc'') = \Pi''(loc''), A_k = A'' \bullet \\ \Bigg(\Big(\langle \mathbb{C}_k[\texttt{invoke}\ loc''], S_k, \Pi_k, A_k\rangle \hookrightarrow^* \langle \mathbb{C}_k[skip], S'_k, \Pi'_k, A'_k\rangle\Big) \\ \Rightarrow \Big(([\![(P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k} \Rightarrow [\![(Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k,S'_k}) \wedge \\ (\Delta(S_k, S'_k) \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k} \cup [\![\varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k})\Big)\Bigg)\end{array}\right) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ ^{loc''}/_{var_{h_1}},\ ^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$$\Longrightarrow \langle \text{by } 18.2, 18.3, 18.4, \text{axiom 17 and definition 15} \rangle$$

$$\left[\begin{array}{c} \left([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'',S'} \right) \wedge \\ \left(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \right) \end{array} \right]$$

$$\Longrightarrow \langle \text{by construction of } S'' \rangle$$

$$\left[\begin{array}{c} \left([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'',S'} \right) \wedge \\ \left(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \right) \wedge \\ \left([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \Leftrightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \right) \wedge \\ \left([\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'} \Leftrightarrow [\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'',S'} \right) \wedge \\ \left(dom(S) \subseteq dom(S'') \right) \wedge \\ \left([\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} = [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \right) \end{array} \right]$$

$$\Longrightarrow \langle \text{by predicate calculus, definition of } \Delta \text{ and set theory} \rangle$$

$$\left[\begin{array}{c} \left([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \Rightarrow [\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'} \right) \wedge \\ \left(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \right) \end{array} \right]$$

$$\Longrightarrow \langle \text{by predicate calculus and set theory} \rangle$$

$$\left[\begin{array}{c} \left([\![(P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \Rightarrow [\![(Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'} \right) \wedge \\ \left(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \cup [\![\varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \right) \wedge \\ \left([\![(P_{e_H[\rho(var_i)/var_i]_{i=1}^n} \wedge P_e)]\!]_{S} \Rightarrow [\![(Q_{e_H[\rho(var_i)/var_i]_{i=1}^n} \vee Q_e)]\!]_{S,S'} \right) \wedge \\ \left(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \cup [\![\varepsilon_e]\!]_{S} \right) \end{array} \right] \blacksquare$$

Using theorem 18 it is easy to demonstrate the soundness of the proof rules for PtolemyRely. For the (PTOLEMYRELY-INVOKE$_S$) rule, according to 5.12 we need to prove the following:

Corollary 19 (PtolemyRely Invoke Rule Soundness)**.**

$\forall\ S, \Pi, A : valid(S, \Pi, A)$ •

$$\left[\begin{array}{c} \Big((\textbf{thunk}\ ev) = \mathit{\Pi}(loc)\Big) \wedge \\ \Big((\textbf{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \textbf{relies}\ contract_B\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \textbf{requires}\ P_{e_B}\ \textbf{modifies}\ \varepsilon_{e_B}\ \textbf{ensures}\ Q_{e_B}\Big) \wedge \\ \Big(contract_H = \ \textbf{requires}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\ \textbf{assumes}\ A_{e_H}\ \textbf{ensures}\ Q_{e_H}\Big) \wedge \\ \Big(\textbf{eClosure}(H, c, \rho) = S(loc)\Big) \end{array}\right] \begin{array}{c} (19.1) \\ (19.2) \\ (19.3) \\ (19.4) \\ (19.5) \end{array}$$

$\Longrightarrow$

$$\left[\begin{array}{c} \Big(\langle \mathbb{C}[\textbf{invoke}\ loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big) \\ \Longrightarrow \Big(([\![(P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![(Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'}) \wedge \\ (\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \cup [\![\varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^n}]\!]_S)\Big) \end{array}\right] \quad (19.6)$$

Proof: [PtolemyRely Invoke Rule Soundness]

The proof follows immediately from Theorem 18 using $(P_c, Q_c, \varepsilon_c) = (P_{eB}, Q_{eB}, \varepsilon_{eB})$, *which is satisfied by* c*, according to axiom 16* ∎

For the (PTOLEMYRELY-ANNOUNCE$_S$) rule, according to 5.11 we need to prove the following:

Corollary 20 (PtolemyRely Announce Rule Soundness)**.**

$\forall\ S, \Pi, A : valid(S, \Pi, A)$ •

$$\left[\begin{array}{c} \Big((\textbf{\textit{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \textbf{\textit{relies}}\ contract_B\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \textbf{\textit{requires}}\ P_{e_B}\ \textbf{\textit{modifies}}\ \varepsilon_{e_B}\ \textbf{\textit{ensures}}\ Q_{e_B}\Big) \wedge \\ \Big(contract_H = \textbf{\textit{requires}}\ P_{e_H}\ \textbf{\textit{modifies}}\ \varepsilon_{e_H}\ \textbf{\textit{assumes}}\ A_{e_H}\ \textbf{\textit{ensures}}\ Q_{e_H}\Big) \wedge \\ \Big([\![e_1]\!]_S = v_1, \ \ \ldots, \ \ [\![e_n]\!]_S = v_n\Big) \wedge \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \end{array}\right] \begin{array}{c} (20.1) \\ (20.2) \\ (20.3) \\ (20.4) \\ (20.5) \end{array}$$

$$\Longrightarrow$$

$$\left[\begin{array}{c} \Big(\langle \mathbb{C}[\textbf{\textit{announce}}\ ev(e_1, ..., e_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \\ \Longrightarrow \Big(([\![(P_{e_H[v_i/var_i]_{i=1}^n} \wedge P_c)]\!]_S \Rightarrow [\![(Q_{e_H[v_i/var_i]_{i=1}^n} \vee Q_c)]\!]_{S,S'}) \wedge \\ (\Delta(S, S') \subseteq [\![\varepsilon_{e_H[v_i/var_i]_{i=1}^n}]\!]_S \cup [\![\varepsilon_c]\!]_S)\Big) \end{array}\right] \begin{array}{c} (20.6) \\ (20.7) \\ \ \end{array}$$

Proof: [PtolemyRely Announce Rule Soundness]

$\langle$by 20.6$\rangle$

$$\Big[\langle \mathbb{C}[\textbf{announce}\ ev(e_1, ..., e_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big]$$

$\Longrightarrow$ $\langle$by evaluation order imposed by evaluation contexts and 20.4 $\rangle$

$$\left[\begin{array}{c} \Big(\langle \mathbb{C}[\textbf{announce}\ ev(e_1, ..., e_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \\ \langle \mathbb{C}[\textbf{announce}\ ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[\textbf{announce}\ ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A\rangle\Big) \end{array}\right]$$

$\Longrightarrow$ $\langle$by (PTOLEMYRELY-ANNOUNCE$_S$) rule $\rangle$

$$\left[\begin{array}{c}\Big(H = handlersOf(ev, A)\Big) \;\wedge\; \Big(\;\rho = \{var_i \mapsto v_i\}_{i=1}^{n}\Big) \wedge \\ \Big(loc'' \notin dom(S)\Big) \wedge \\ \Big(S'' = S \uplus (loc'' \mapsto \texttt{eClosure}(H, c, \rho))\Big) \wedge \\ \Big(\Pi'' = \Pi \uplus \{loc'' \;:\; var\; \texttt{thunk}\; ev\}\Big) \wedge \\ \Big(A'' = A\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle \hookrightarrow^{*} \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by 20.1, 20.2, 20.3, 20.5 ⟩

$$\left[\begin{array}{c}\Big((\texttt{thunk}\; ev) = \Pi''(loc'')\Big) \wedge \\ \Big((\texttt{event}\; ev\; \{t_1\; var_1, ..., t_n\; var_n\; \texttt{relies}\; contract_B\; contract_H\}) \in CT\Big) \wedge \\ \Big(contract_B = \texttt{requires}\; P_{e_B}\; \texttt{modifies}\; \varepsilon_{e_B}\; \texttt{ensures}\; Q_{e_B}\Big) \wedge \\ \Big(contract_H = \;\texttt{requires}\; P_{e_H}\; \texttt{modifies}\; \varepsilon_{e_H}\; \texttt{assumes}\; A_{e_H}\; \texttt{ensures}\; Q_{e_H}\Big) \wedge \\ \Big(\texttt{eClosure}(H, c, \rho) = S''(loc'')\Big) \wedge \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle \hookrightarrow^{*} \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by theorem 18⟩

$$\left[\begin{array}{c}\Big(\llbracket (P_{e_H} \wedge P_{e_B})_{[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S''} \Rightarrow \llbracket (Q_{e_H} \vee Q_{e_B})_{[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S'',S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S''} \cup \llbracket \varepsilon_{e_B[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S''}\Big) \wedge \\ \Big(\llbracket (P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \wedge P_c) \rrbracket_{S''} \Rightarrow \llbracket (Q_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \vee Q_c) \rrbracket_{S'',S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S''} \cup \llbracket \varepsilon_c \rrbracket_{S''}\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by predicate calculus⟩

$$\left[\begin{array}{c}\Big(\llbracket (P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \wedge P_c) \rrbracket_{S''} \Rightarrow \llbracket (Q_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \vee Q_c) \rrbracket_{S'',S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}} \rrbracket_{S''} \cup \llbracket \varepsilon_c \rrbracket_{S''}\Big)\end{array}\right]$$

$$\Longrightarrow \langle \text{by construction of } S'' \rangle$$

$$\left[\begin{array}{c} \Big([\![(P_{e_H[\rho(var_i)/var_i]_{i=1}^n} \wedge P_c)]\!]_{S''} \Rightarrow [\![(Q_{e_H[\rho(var_i)/var_i]_{i=1}^n} \vee Q_c)]\!]_{S'',S'} \Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \cup [\![\varepsilon_c]\!]_{S''} \Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Leftrightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Big) \wedge \\ \Big([\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'} \Leftrightarrow [\![Q_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'',S'} \Big) \wedge \\ \Big([\![P_c]\!]_S \Leftrightarrow [\![P_c]\!]_{S''} \Big) \wedge \\ [\![Q_c]\!]_{S,S'} \Leftrightarrow [\![Q_c]\!]_{S'',S'} \Big) \wedge \\ \Big(dom(S) \subseteq dom(S'') \Big) \wedge \\ \Big([\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S = [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Big) \wedge \\ \Big([\![\varepsilon_c]\!]_S = [\![\varepsilon_c]\!]_{S''} \Big) \end{array}\right]$$

$$\Longrightarrow \langle \text{by predicate calculus, definition of } \Delta \text{ and set theory} \rangle$$

$$\left[\begin{array}{c} \Big([\![(P_{e_H[\rho(var_i)/var_i]_{i=1}^n} \wedge P_c)]\!]_S \Rightarrow [\![(Q_{e_H[\rho(var_i)/var_i]_{i=1}^n} \vee Q_c)]\!]_{S,S'} \Big) \wedge \\ \Big(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \cup [\![\varepsilon_c]\!]_S \Big) \end{array}\right]$$

$$\Longrightarrow \langle \text{by } \rho(var_i) = v_i \rangle$$

$$\left[\begin{array}{c} \Big([\![(P_{e_H[v_i/var_i]_{i=1}^n} \wedge P_c)]\!]_S \Rightarrow [\![(Q_{e_H[v_i/var_i]_{i=1}^n} \vee Q_c)]\!]_{S,S'} \Big) \wedge \\ \Big(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[v_i/var_i]_{i=1}^n}]\!]_S \cup [\![\varepsilon_c]\!]_S \Big) \end{array}\right] \blacksquare$$

6.1.2 Full Delivery II

In full-delivery the semantics of an event announcement is that every registered handler for the announced event, if any, is executed sequentially. It is similar to a **`while`** loop whose body executes

a different handler in every iteration. The dynamic semantics rules are shown in Figure 6.3.

(FULLDELIVERY-ANNOUNCE$_S$)

$$\frac{\begin{array}{c}(\texttt{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT,\\ H = handlersOf(ev, A),\ \rho = \{var_i \mapsto v_i\}_{i=1}^{n},\ loc \notin dom(S),\\ S' = S \uplus (loc \mapsto \texttt{eClosure}(H, \rho)),\\ \Pi' = \Pi \uplus \{loc\ :\ var\ \texttt{thunk}\ ev\},\\ A' = A\end{array}}{CT \vdash \langle \mathbb{C}[\texttt{announce}\ ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[\texttt{invoke}\ loc], S', \Pi', A'\rangle}$$

(FULLDELIVERY-INVOKEDONE$_S$)

$$\frac{\texttt{eClosure}(\bullet, \rho) = S(loc)}{CT \vdash \langle \mathbb{C}[\texttt{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[\texttt{skip}], S, \Pi, A\rangle}$$

(FULLDELIVERY-INVOKE$_S$)

$$\frac{\begin{array}{c}(\texttt{thunk}\ ev) = \Pi(loc), \qquad (\texttt{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT,\\ \texttt{eClosure}(H, \rho) = S(loc),\ \ H = \langle loc_h, m_h\rangle + H',\\ [C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h),\ \ loc' \notin dom(S),\\ S' = S \uplus (loc' \mapsto \texttt{eClosure}(H', \rho)),\\ \Pi' = \Pi \uplus \{loc'\ :\ var\ \texttt{thunk}\ ev\},\\ A' = A\end{array}}{CT \vdash \langle \mathbb{C}[\texttt{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}\big[c_h[(^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^{n},\ ^{loc_h}/_{this}]\big];\ \texttt{invoke}\ loc'], S', \Pi', A'\rangle}$$

(FULLDELIVERY-SKIP)$_S$

$$\frac{}{CT \vdash \langle \mathbb{C}[skip; c], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[c], S, \Pi, A\rangle}$$

Figure 6.3: Full Delivery Semantics.

As all registered handlers are mandatorily executed, no surface statement for next-handler invocation (**invoke** or **proceed**) is required. There is no announced-code to proceed to, so event closures do not consider it, and only includes the list of handler methods and an environment representing the context variables: $ec ::= \texttt{eClosure}(H, \rho)$. The (FULLDELIVERY-ANNOUNCE$_S$) rule creates a closure that includes the list of handlers for the event and an environment mapping the context variables to its actual values. Then it stores that closure in a fresh location loc and starts running this closure by invoking that location. The (FULLDELIVERY-INVOKEDONE$_S$)

rule applies when there are no more handlers in the closure; it just skips. The (FULLDELIVERY-INVOKE$_S$) rule evaluates the invocation (**invoke**) of a location into the execution of the body c_h of the first handler in the closure at that location, followed by another **invoke** statement to execute the rest of the handlers. Once the first handler is taken, the rest of the closure (remaining handlers and same environment) is stored into a fresh location loc'. Every reference to a context field $var_{h_1}.var_i$ is substituted by the corresponding field in the environment $\rho(var_i)$. It also substitutes any reference to **this** by a reference to the observer object loc_h containing the handler method.

We assume *handler abstraction* (5.5, 5.6): handlers satisfy their invariant: $c_h \sqsupseteq (P_{e_H}, P_{e_H}, \varepsilon_{e_H})$.

Axiom 21 (Handlers satisfy invariant)**.**

$$\forall\ S, \Pi, A : valid(S, \Pi, A) \bullet$$
$$\left[\begin{array}{c} \Big(\langle loc_h, m_h\rangle \in handlersOf(ev)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h),\ \ (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\textbf{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_H =\ \textbf{invariant}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\Big) \wedge \\ \Big((\textbf{thunk}\ ev) = \Pi(loc)\Big)\ \wedge\ \Big(\textbf{eClosure}(H, \rho) = S(loc)\Big) \end{array}\right]$$

$$\Longrightarrow \left[\begin{array}{c} \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^{n},\ {}^{loc_h}/_{this}]\big], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \\ \Rightarrow \Big(([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'}) \\ \wedge(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S)\Big) \end{array}\right]$$

Each handler should keep the event invariant and respect the event frame. As handlers are executed in sequence then the frame (ε_{eH}) must be immune [8] from itself under the invariant (P_{eH}). That allows the sequential composition of two or more handlers to also respect the frame condition.

Axiom 22 (Frame immunity)**.**

$$\left[\begin{array}{c}\Big((\textbf{\texttt{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_H = \textbf{\texttt{invariant}}\ P_{e_H}\ \textbf{\texttt{modifies}}\ \varepsilon_{e_H}\Big)\end{array}\right]$$

$$\Longrightarrow \left[\forall S \bullet \Big([\![P_{eH}]\!]_S \Rightarrow (footprint([\![\varepsilon_{eH}]\!]_S) \cap [\![\varepsilon_{eH}]\!]_S = \emptyset)\Big)\right]$$

Using these axioms we can prove the following **`invoke`** reasoning theorem.

Theorem 23 (Invoke Reasoning)**.** *For all valid states (S, Π, A) the following holds: given an event, ev, and an closure for it, stored at a location loc, **`eClosure`**$(H, \rho) = S(loc)$, then an **`invoke`** statement of the form **`invoke`** loc refines the event handlers invariant $(P_{eH}, P_{eH}, \varepsilon_{eH})$. That is:*

$$\left[\begin{array}{c}\Big((\textbf{\texttt{thunk}}\ ev) = \mathit{\Pi}(loc)\Big) \wedge \\ \Big((\textbf{\texttt{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT\Big) \wedge \\ \Big(contract_H = \textbf{\texttt{invariant}}\ P_{e_H}\ \textbf{\texttt{modifies}}\ \varepsilon_{e_H}\Big) \wedge \\ \Big(\textbf{\texttt{eClosure}}(H, \rho) = S(loc)\Big)\end{array}\right] \begin{array}{r}(23.1)\\(23.2)\\(23.3)\\(23.4)\end{array}$$

$$\Longrightarrow$$

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\textbf{\texttt{invoke}}\ loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big) \\ \Rightarrow \Big(([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'}) \wedge \\ (\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S)\Big)\end{array}\right] \quad (23.5)$$

Proof: [by induction over the length of the list of handlers $|H|$]

Case 1. $[|H| = 0 \ \ i.e. \ H = \bullet]$

$$\langle \text{by 23.5} \rangle$$
$$\Big[\langle \mathbb{C}[\texttt{invoke}\, loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle \Big]$$

$$\Longrightarrow \ \langle \text{by hypothesis 23.4 and case hypothesis} \rangle$$
$$\left[\begin{array}{c} \Big(\texttt{eClosure}(\bullet, \rho) = S(loc) \Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\, loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle \Big) \end{array} \right]$$

$$\Longrightarrow \ \langle \text{by (FULLDELIVERY-INVOKEDONE}_S\text{) semantic rule and transitivity of } \hookrightarrow \rangle$$
$$\left[\begin{array}{c} \Big(S' = S \Big) \ \wedge \ \Big(\Pi' = \Pi \Big) \ \wedge \ \Big(A' = A \Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\, loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[skip], S', \Pi', A'\rangle \Big) \end{array} \right]$$

$$\Longrightarrow \ \langle \text{by } S' = S \text{ and so } \Delta(S, S') = \emptyset \ \rangle$$
$$\left[\begin{array}{c} \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'} \Big) \wedge \\ \Big(\Delta(S, S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Big) \end{array} \right] \ \blacksquare$$

Case 2. [induction: $H = \langle loc_h, m_h\rangle + H''$, assume for H'' and prove for H]

$$\langle \text{by 23.5} \rangle$$
$$\Big[\langle \mathbb{C}[\texttt{invoke}\, loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle \Big]$$

$$\Longrightarrow \ \langle \text{by case hypothesis, 23.1, 23.4 and } \texttt{eClosure} \text{ well-formedness} \ \rangle$$
$$\left[\begin{array}{c} \Big(H = \langle loc_h, m_h\rangle + H'' \Big) \wedge \\ \Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A) \Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\, loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle \Big) \end{array} \right]$$

$\Longrightarrow$ $\langle$by 23.1, 23.2, 23.4, (FULLDELIVERY-INVOKE$_S$) semantic rule and transitivity of $\hookrightarrow\rangle$

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h),\ \ (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big(loc'' \notin dom(S)\Big) \wedge \\ \Big(S'' = S \uplus (loc'' \mapsto \mathbf{eClosure}(H'', \rho))\Big) \wedge \\ \Big(\Pi'' = \Pi \uplus \{loc''\ :\ var\ \mathbf{thunk}\ ev\}\Big) \wedge \\ \Big(A'' = A\Big) \wedge \\ \Big(\langle \mathbb{C}[\mathbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc_h}/_{this}];\mathbf{invoke}\ loc''\big], S'', \Pi'', A''\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc_h}/_{this}];\mathbf{invoke}\ loc''\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by construction of loc'', S'', Π'' and $H''\rangle$

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h),\ \ (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\mathbf{thunk}\ ev) = \Pi''(loc'')\Big) \wedge \\ \Big(\mathbf{eClosure}(H'', \rho) = S''(loc'')\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc_h}/_{this}];\mathbf{invoke}\ loc''\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ $\langle$by sequence semantics rule$\rangle$

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big((\mathbf{thunk}\ ev) = \Pi''(loc'')\Big) \wedge \\ \Big(\mathbf{eClosure}(H'', \rho) = S''(loc'')\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S''', \Pi''', A'''\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[\mathbf{invoke}\ loc''], S''', \Pi''', A'''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$$\Longrightarrow \langle \text{by handler's invariant axiom (21)} \rangle$$

$$\left[\begin{array}{c} \Big((\mathtt{thunk}\ ev) = \Pi''(loc'')\Big) \wedge \\ \Big(\mathtt{eClosure}(H'', \rho) = S''(loc'')\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''}\Big) \wedge \\ \Big(\Delta(S'', S''') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big) \wedge \\ \Big(\langle \mathbb{C}[\mathtt{invoke}\ loc''], S''', \Pi''', A'''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \end{array}\right]$$

$$\Longrightarrow \langle \text{by induction hypothesis over } \mathtt{invoke}\ loc'' \rangle$$

$$\left[\begin{array}{c} \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''}\Big) \wedge \\ \Big(\Delta(S'', S''') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''})\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'}\Big) \wedge \\ \Big(\Delta(S''', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''}\Big) \end{array}\right]$$

$$\Longrightarrow \langle \text{by Immunity axiom (22) and set theory } (A \cup A = A) \rangle$$

$$\left[\begin{array}{c} \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''}\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]v}]\!]_{S'''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big) \end{array}\right]$$

$$\Longrightarrow \langle \text{by construction of } S'' \rangle$$

$$\left[\begin{array}{c} \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''}\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} \Leftrightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big) \wedge \\ \Big(dom(S) \subseteq dom(S'')\Big) \wedge \\ \Big([\![(\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S} = [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big) \end{array}\right]$$

$\Longrightarrow$ ⟨by predicate calculus and definition of Δ ⟩

$$\left[\begin{array}{c}\left(\llbracket P_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow P_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'}\right) \wedge \\ \left(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S\right)\end{array}\right] \blacksquare$$

The statement and demonstration of the soundness of the *full-delivery* (ANNOUNCE) proof rule 5.7 follows.

Corollary 24 (Full-Delivery Announce Rule Soundness)**.**

$\forall\ S, \Pi, A : valid(S, \Pi, A)$ •

$$\left[\begin{array}{c}\left((\textbf{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ contract_H\}) \in CT\right) \wedge \\ \left(contract_H = \textbf{invariant}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\right) \wedge \\ \left(\llbracket e_1\rrbracket_S = v_1\right)\ \wedge\ \ldots\ \wedge\ \left(\llbracket e_n\rrbracket_S = v_n\right)\end{array}\right] \qquad \begin{array}{r}(24.1)\\(24.2)\\(24.3)\end{array}$$

$\Longrightarrow$

$$\left[\begin{array}{c}\left(\langle \mathbb{C}[\textbf{announce}\ ev(e_1, ..., e_n)], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right) \\ \Longrightarrow \left((\llbracket P_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket P_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_{S'}) \right. \\ \left. (\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_S)\right)\end{array}\right] \qquad (24.4)$$

Proof: [Full-Delivery Announce Rule Soundness]

⟨by 24.4 ⟩

$$\left[\langle \mathbb{C}[\textbf{announce}\ ev(e_1, ..., e_n)], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right]$$

$\Longrightarrow$ ⟨by evaluation order imposed by evaluation contexts and 24.3 ⟩

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\texttt{announce}\; ev(e_1,...,e_n)], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[\texttt{announce}\; ev(v_1,...,v_n)], S, \Pi, A\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{announce}\; ev(v_1,...,v_n)], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by (FULLDELIVERY-ANNOUNCE$_S$) rule and transitivity of $\hookrightarrow$⟩

$$\left[\begin{array}{c}\Big(H = handlersOf(ev, A)\Big) \;\wedge\; \Big(\rho = \{var_i \mapsto v_i\}_{i=1}^{n}\Big) \wedge \\ \Big(\; loc'' \notin dom(S)\Big) \wedge \\ \Big(S'' = S \uplus (loc'' \mapsto \texttt{eClosure}(H, \rho))\Big) \wedge \\ \Big(\Pi'' = \Pi \uplus \{loc'' \;:\; var\; \texttt{thunk}\; ev\}\Big) \wedge \\ \Big(A'' = A\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{announce}\; ev(v_1,...,v_n)], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \wedge\end{array}\right]$$

$\Longrightarrow$ ⟨by theorem 23⟩

$$\left[\begin{array}{c}\Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''}\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by construction of S''⟩

$$\left[\begin{array}{c}\Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''} \Rightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''}\Big) \wedge \\ \Big([\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S} \Leftrightarrow [\![P_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''}\Big) \wedge \\ \Big(dom(S) \subseteq dom(S'')\Big) \wedge \\ \Big([\![(\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S} = [\![\varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^{n}}]\!]_{S''}\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by predicate calculus and definition of Δ⟩

$$\left[\begin{array}{c}\left(\llbracket P_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket P_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'}\right) \wedge \\ \left(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_S\right)\end{array}\right]$$

$$\Longrightarrow \langle \text{by } \rho(var_i) = v_i \rangle$$

$$\left[\begin{array}{c}\left(\llbracket P_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_S \Rightarrow \llbracket P_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_{S'}\right) \wedge \\ \left(\Delta(S,S') \subseteq \llbracket \varepsilon_{e_H[v_i/var_i]_{i=1}^n}\rrbracket_S\right)\end{array}\right] \blacksquare$$

6.1.3 Single Delivery II: Non-Modular Individual Refinement

The semantics for this case is the same as that for PtolemyRely (Figure 6.2). The difference lies in the specification and verification features. In the non-modular scenario the event declaration does not include specifications for the handlers and for the announced-code. Each event announcement is reasoned about in a case-by-case non-modular fashion, as described in section 5.4.2.4.

The following theorem formalizes how to reason about **`invoke`** statements. In this context the function *varsOf*(ev, CT) is defined as follows:

$$(t_1\ var_1, ..., t_n\ var_n) \equiv \textit{varsOf}(ev, CT) \Leftrightarrow (\textbf{event}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \}) \in CT \quad (6.2)$$

Theorem 25 (Individual Refinement Invoke Reasoning)**.** *For all valid states* (S, Π, A)*, given an event,* ev*, and a closure for it, stored at a location* loc*,* **`eClosure`**$(H, c, \rho) = S(loc)$*, the following holds: if the expression* c *satisfies a specification* $(P_c, Q_c, \varepsilon_c)$ *and if every handler for that event, proceed-composed with this specification, also refines it, then a statement of the form* **`invoke`** loc

also refines this specification. That is:

$$\left[\begin{array}{c} \Big((\textbf{\textit{thunk}}\ ev) = \Pi(loc)\Big) \wedge \\ \Big((\textbf{\textit{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\ \}) \in CT\Big) \wedge \\ \Big(\textbf{\textit{eClosure}}(H, c, \rho) = S(loc)\Big) \wedge \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)_{[\rho(var_i)/var_i]_{i=1}^{n}}\Big) \wedge \\ \Big(\forall \langle loc_h, m_h\rangle \in handlersOf(ev, A) \bullet [\langle loc_h, m_h\rangle \circledcirc (P_c, Q_c, \varepsilon_c)] \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \end{array}\right] \begin{array}{r} (25.1) \\ (25.2) \\ (25.3) \\ (25.4) \\ (25.5) \end{array}$$

$\Longrightarrow$

$$\left[\begin{array}{c} \Big(\langle \mathbb{C}[\textbf{\textit{invoke}}\ loc], S, \Pi, A\rangle \hookrightarrow^{*} \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \\ \Rightarrow \Big((\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^{n}}\rrbracket_S \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^{n}}\rrbracket_{S,S'}) \wedge \\ (\Delta(S, S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^{n}}\rrbracket_S)\Big) \end{array}\right] \quad (25.6)$$

Proof: [by induction over the length of the list of handlers $|H|$]

Case 1. [$|H| = 0\ \ i.e.\ H = \bullet$]

⟨by 25.6⟩

$$\Big[\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow^{*} \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big]$$

⟨by hypothesis 25.1, 25.3 and case hypothesis⟩

$$\left[\begin{array}{c} \Big(\textbf{eClosure}(\bullet, c, \rho) = S(loc)\Big) \wedge \\ \Big(\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow^{*} \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \end{array}\right]$$

$\Longrightarrow$ ⟨by (PTOLEMYRELY-INVOKEDONE$_S$) semantic rule and transitivity of $\hookrightarrow$ ⟩

$$\left[\begin{array}{c}\left(\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}[c], S, \Pi, A\rangle\right) \wedge \\ \left(\langle \mathbb{C}[c], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by 25.4: $c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)_{[\rho(var_i)/var_i]_{i=1}^n}$ and definition 14 $\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n} \rrbracket_S \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n} \rrbracket_{S,S'}\right) \wedge \\ \left(\Delta(S, S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n} \rrbracket_S\right)\end{array}\right] \blacksquare$$

Case 2. [induction: $H = \langle loc_h, m_h\rangle + H''$, assume for H'' and prove for H]

$\langle$by 25.6$\rangle$

$$\left[\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right]$$

$\Longrightarrow$ $\langle$by case hypothesis and 25.1, 25.3 and **eClosure** well-formedness $\rangle$

$$\left[\begin{array}{c}\left(H = \langle loc_h, m_h\rangle + H''\right) \wedge \\ \left(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\right) \wedge \\ \left(\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by 25.1, 25.2, 25.3, (PTOLEMYRELY-INVOKE$_S$) semantic rule and transitivity of $\hookrightarrow$ $\rangle$

$$\left[\begin{array}{c}\left(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\right) \wedge \\ \left([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\right) \wedge \\ \left(loc'' \notin dom(S)\right) \wedge \\ \left(S'' = S \uplus (loc'' \mapsto \textbf{eClosure}(H'', c, \rho))\right) \wedge \\ \left(\Pi'' = \Pi \uplus \{loc'' : var\ \textbf{thunk}\ ev\}\right) \wedge \\ \left(A'' = A\right) \wedge \\ \left(\langle \mathbb{C}[\textbf{invoke}\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{C}\left[c_h[(\tfrac{\rho(var_i)/}{var_{h_1}.var_i})_{i=1}^n, \tfrac{loc''/}{var_{h_1}}, \tfrac{loc_h/}{this}]\right], S'', \Pi'', A''\rangle\right) \wedge \\ \left(\langle \mathbb{C}\left[c_h[(\tfrac{\rho(var_i)/}{var_{h_1}.var_i})_{i=1}^n, \tfrac{loc''/}{var_{h_1}}, \tfrac{loc_h/}{this}]\right], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right)\end{array}\right]$$

$\Longrightarrow$ ⟨by construction of loc'', S'', Π'' and H''⟩

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev, A)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\texttt{thunk}\ ev) = \Pi''(loc'')\Big) \wedge \\ \Big(\texttt{eClosure}(H'', c, \rho) = S''(loc'')\Big) \wedge \\ \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc''}/_{var_{h_1}},\ {}^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by the inductive hypothesis applied to every [**invoke** loc''] in c_h⟩

$$\left[\begin{array}{c}\Big(\langle loc_h, m_h\rangle \in handlersOf(ev)\Big) \wedge \\ \Big([C_h.F_h] = S(loc_h), (t_h\ m_h(t_{h_1}\ var_{h_1})\{c_h\}) = methodBody(C_h, m_h)\Big) \wedge \\ \Big((\texttt{thunk}\ ev) = \Pi''(loc'')\Big) \wedge \\ \Big(\texttt{eClosure}(H'', c, \rho) = S''(loc'')\Big) \wedge \\ \left(\begin{array}{c}\forall S_k, \Pi_k, A_k, \mathbb{C}_k\ s.t.\ S_k(loc'') = S''(loc''), \Pi_k(loc'') = \Pi''(loc''), A_k = A'' \bullet \\ \Bigg(\Big(\langle \mathbb{C}_k[\texttt{invoke}\ loc''], S_k, \Pi_k, A_k\rangle \hookrightarrow^* \langle \mathbb{C}_k[skip], S'_k, \Pi'_k, A'_k\rangle\Big) \\ \Rightarrow \Big(([\![P_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k} \Rightarrow [\![Q_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k,S'_k}) \wedge \\ (\Delta(S_k, S'_k) \subseteq [\![\varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S_k})\Big)\Bigg)\end{array}\right) \\ \Big(\langle \mathbb{C}\big[c_h[({}^{\rho(var_i)}/_{var_{h_1}.var_i})_{i=1}^n,\ {}^{loc''}/_{var_{h_1}},\ {}^{loc_h}/_{this}]\big], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by 25.5 and definition 15⟩

$$\left[\begin{array}{c}\Big([\![P_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''} \Rightarrow [\![Q_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S'',S'}\Big) \wedge \\ \Big(\Delta(S'', S') \subseteq [\![\varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S''}\Big)\end{array}\right]$$

$\Longrightarrow$ ⟨by construction of S''⟩

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''} \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'',S'}\right) \wedge \\ \left(\Delta(S'', S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right) \wedge \\ \left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} \Leftrightarrow \llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right) \wedge \\ \left(\llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'} \Leftrightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'',S'}\right) \wedge \\ \left(dom(S) \subseteq dom(S'')\right) \wedge \\ \left(\llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} = \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by predicate calculus, definition of Δ and set theory$\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}\right) \wedge \\ \left(\Delta(S, S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S}\right)\end{array}\right] \blacksquare$$

Using theorem 25 it is easy to demonstrate the soundness of the proof rules for the Non-Modular Individual Refinement scenario. For the (INVOKE) rule, according to 5.24 we need to prove the following:

Corollary 26 (Individual Refinement Invoke Rule Soundness)**.**

$\forall\, S, \Pi, A : valid(S, \Pi, A) \bullet$

$$\left[\begin{array}{c}\left((\textbf{\texttt{thunk}}\; ev) = \mathit{\Pi}(loc)\right) \wedge \quad (26.1)\\ \left((\textbf{\texttt{event}}\; ev\; \{t_1\; var_1, ..., t_n\; var_n\}) \in CT\right) \wedge \quad (26.2)\\ \left(\textbf{\texttt{eClosure}}(H, c, \rho) = S(loc)\right) \wedge \quad (26.3)\\ \left(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)_{[\rho(var_i)/var_i]_{i=1}^n}\right) \wedge \quad (26.4)\\ \left(\forall \langle loc_h, m_h\rangle \in handlersOf(ev, A) \bullet [\langle loc_h, m_h\rangle \circledcirc (P_c, Q_c, \varepsilon_c)] \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\right) \quad (26.5)\end{array}\right]$$

$\Longrightarrow$

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\textbf{\texttt{invoke}}\ loc], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big) \\ \Longrightarrow \Big(([\![P_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![Q_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_{S,S'}) \wedge \\ (\Delta(S,S') \subseteq [\![\varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}]\!]_S)\Big)\end{array}\right] \tag{26.6}$$

Proof: [Individual Refinement Invoke Rule Soundness]

The proof follows immediately from Theorem 18. ∎

For the (ANNOUNCE) rule, according to 5.23 we need to prove the following:

Corollary 27 (Individual Refinement Announce Rule Soundness)**.**

$\forall\ S, \Pi, A : valid(S, \Pi, A) \bullet$

$$\left[\begin{array}{c}\Big((\textbf{\texttt{event}}\ ev\ \{t_1\ var_1, ..., t_n\ var_n\}) \in CT\Big) \wedge \quad (27.1) \\ \Big([\![e_1]\!]_S = v_1\Big) \wedge\ \ldots\ \wedge \Big([\![e_n]\!]_S = v_n\Big) \wedge \quad (27.2) \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)_{[v_i/var_i]_{i=1}^n}\Big) \wedge \quad (27.3) \\ \Big(\forall \langle loc_h, m_h\rangle \in handlersOf(ev, A) \bullet [\langle loc_h, m_h\rangle \circledcirc (P_c, Q_c, \varepsilon_c)] \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \quad (27.4)\end{array}\right]$$

$\Longrightarrow$

$$\left[\begin{array}{c}\Big(\langle \mathbb{C}[\textbf{\texttt{announce}}\ ev(e_1, ..., e_n)\{c\}], S, \mathit{\Pi}, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \mathit{\Pi}', A'\rangle\Big) \\ \Longrightarrow \Big(([\![P_{c[v_i/var_i]_{i=1}^n}]\!]_S \Rightarrow [\![Q_{c[v_i/var_i]_{i=1}^n}]\!]_{S,S'}) \\ \wedge (\Delta(S,S') \subseteq [\![\varepsilon_{c[v_i/var_i]_{i=1}^n}]\!]_S)\Big)\end{array}\right] \tag{27.5}$$

Proof: [Individual Refinement Announce Rule Soundness]

$\langle$by 27.5 $\rangle$

$$\left[\langle \mathbb{C}[\texttt{announce}\; ev(e_1, ..., e_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\right]$$

$\Longrightarrow$ $\langle$by evaluation order imposed by evaluation contexts and 27.2 $\rangle$

$$\left[\begin{array}{c} \Big(\langle \mathbb{C}[\texttt{announce}\; ev(e_1, ..., e_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \\ \langle \mathbb{C}[\texttt{announce}\; ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{announce}\; ev(v_1, ..., v_n)\{c\}], S, \Pi, A\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A\rangle\Big) \end{array}\right]$$

$\Longrightarrow$ $\langle$by (PTOLEMYRELY-ANNOUNCE$_S$) rule $\rangle$

$$\left[\begin{array}{c} \Big(H = handlersOf(ev, A)\Big) \;\wedge\; \Big(\rho = \{var_i \mapsto v_i\}_{i=1}^{n}\Big) \wedge \\ \Big(loc'' \notin dom(S)\Big) \wedge \\ \Big(S'' = S \uplus (loc'' \mapsto \texttt{eClosure}(H, c, \rho))\Big) \wedge \\ \Big(\Pi'' = \Pi \uplus \{loc'' \;:\; var\; \texttt{thunk}\; ev\}\Big) \wedge \\ \Big(A'' = A\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \end{array}\right]$$

$\Longrightarrow$ $\langle$by 27.1, 27.3, 27.4 and $\rho(var_i) = v_i\rangle$

$$\left[\begin{array}{c} \Big((\texttt{thunk}\; ev) = \Pi(loc'')\Big) \wedge \\ (\texttt{event}\; ev\; \{t_1\; var_1, ..., t_n\; var_n\}) \in CT\Big) \wedge \\ \Big(\texttt{eClosure}(H, c, \rho) = S''(loc'')\Big) \wedge \\ \Big(c \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)_{[\rho(var_i)/var_i]_{i=1}^{n}}\Big) \wedge \\ \Big(\forall \langle loc_h, m_h\rangle \in handlersOf(ev, A) \bullet [\langle loc_h, m_h\rangle \circledcirc (P_c, Q_c, \varepsilon_c)] \mathrel{\mathring{\sqsupseteq}} (P_c, Q_c, \varepsilon_c)\Big) \wedge \\ \Big(\langle \mathbb{C}[\texttt{invoke}\; loc''], S'', \Pi'', A''\rangle \hookrightarrow^* \langle \mathbb{C}[skip], S', \Pi', A'\rangle\Big) \end{array}\right]$$

$\Longrightarrow$ $\langle$by theorem 25$\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''} \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'',S'}\right) \wedge \\ \left(\Delta(S'', S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by construction of $S''\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''} \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'',S'}\right) \wedge \\ \left(\Delta(S'', S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right) \\ \left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} \Leftrightarrow \llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right) \wedge \\ \left(\llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'} \Leftrightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S'',S'}\right) \wedge \\ \left(dom(S) \subseteq dom(S'')\right) \wedge \\ \left(\llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} = \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S''}\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by predicate calculus and definition of $\Delta\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S} \Rightarrow \llbracket Q_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S,S'}\right) \wedge \\ \left(\Delta(S, S') \subseteq \llbracket \varepsilon_{c[\rho(var_i)/var_i]_{i=1}^n}\rrbracket_{S}\right)\end{array}\right]$$

$\Longrightarrow$ $\langle$by $\rho(var_i) = v_i\rangle$

$$\left[\begin{array}{c}\left(\llbracket P_{c[v_i/var_i]_{i=1}^n}\rrbracket_{S} \Rightarrow \llbracket Q_{c[v_i/var_i]_{i=1}^n}\rrbracket_{S,S'}\right) \wedge \\ \left(\Delta(S, S') \subseteq \llbracket \varepsilon_{c[v_i/var_i]_{i=1}^n}\rrbracket_{S}\right)\end{array}\right] \blacksquare$$

6.2 Reasoning Cases

In this section case studies involving different reasoning scenarios are presented. Programs are manually reasoned about using the proof rules that apply in each case.

Through this section the *Billing* system example will be used to illustrate the reasoning in each scenario. In this system, each bill includes the amount (a) to be paid and the extra charges (c) like taxes. When the base code totals a bill, adding the charges to the principal amount, the corresponding event is announced. This gives registered handlers the chance to do some adjustments, like adding some extra charges. Specifications are written in a JML[37, 38] like notation.

We use one-state preconditions and two-state postconditions. The state corresponds to the program variables (x) and the heap (σ), that maps reference-field pairs to their values. Unprimed expressions ($x, e.f$) correspond to the current state. In postconditions, left-primed expressions (${}'x$, ${}'e.f$) correspond to their value in the pre-state (equivalent to $\backslash old(\cdot)$ in JML). The map σ correspond to the current state of the heap ($e.f \equiv \sigma(e, f)$) and the map ${}'\sigma$ to the pre-state (${}'e.f \equiv {}'\sigma(e, f)$). As commands only alter the post-state, always the *primed* version of the precondition can be assumed in the post-state. This is expressed by the (PROPAGATION) rule below. The assignment axiom guarantees that any assertion that holds in the post-state of an assignment command also holds in the pre-state, substituting every occurrence of the *target* by the assigned *expression*, while keeping the rest of the state unchanged. This is shown in the (ASSIGN) and (UPDATE) rules.

(PROPAGATION)
$$\frac{\{P\}\ S\ \{Q\}[\varepsilon]}{\{P\}\ S\ \{{}'P \wedge Q\}[\varepsilon]}$$

(ASSIGNMENT)
$$\frac{}{\{P\begin{bmatrix} e/x, \\ \overrightarrow{y}/\overrightarrow{{}'y}, \\ \overrightarrow{e_i.f_i}/\overrightarrow{{}'e_i.f_i} \end{bmatrix}\}\ x = e\ \{P\}[\varepsilon]}$$

(UPDATE)
$$\frac{}{\{P\begin{bmatrix} e/x.f, \\ \overrightarrow{y}/\overrightarrow{{}'y}, \\ \overrightarrow{e_i.f_i}/\overrightarrow{{}'e_i.f_i} \end{bmatrix}\}\ x.f = e\ \{P\}[\varepsilon]}$$

```
1  public class Bill {
2    public int amount;
3    public int charges;
4
5    //@ modifies this.amount,this.charges;
6    //@ ensures this.amount==amount && this.charges==charges;
7    public Bill(int amount, int charges) {
8      this.amount = amount;
9      this.charges = charges;
10   }
11
12   //@ modifies \nothing;
13   //@ ensures \result==amount;
14   public int a() { return amount;  }
15
16   //@ ensures this.amount==amount;
17   //@ modifies this.amount;
18   public void setA(int amount) { this.amount = amount; }
19
20   //@ modifies \nothing;
21   //@ ensures \result==charges;
22   public int c() { return charges; }
23
24   //@ modifies this.charges;
25   //@ ensures this.charges==charges;
26   public void setC(int charges) { this.charges = charges; }
27 }
```

Figure 6.4: Bill class

6.2.1 II/EA Full Delivery Reasoning

In *Full Delivery* systems all the registered handlers for an event are invoked sequentially upon the occurrence of the event. This approach is used in C# *delegates* [22, 46], Java *JavaBeans* [61] and others languages. The *Billing* example is presented in a *Full-Delivery* variant of Ptolemy language.

The **event** declaration (lines 15-19) includes the context variable, *bill*, the event invariant and the **modifies** clause.

```
 1 public class Base {
 2   public void main(){
 3     Bill bill=new Bill(100,8);
 4     registerHandlers();
 5     total(bill);
 6   }
 7   /*@ requires bill.c>=0; @*/                   // c ≥ 0
 8   /*@ modifies bill.a, bill.c; @*/              // [a,c]
 9   /*@ ensures bill.a()>=\old(bill.a()); @*/ // a ≥ 'a
10   public void total(Bill bill){
11     announce TotalingEvent(bill);
12     bill.setA(bill.a()+bill.c()); // a = a + c
13   }
14 }
15 public void event TotalingEvent { // (P_eH, P_eH, ε_eH) ≡ (c ≥ 0,  c ≥ 0,  [c])
16   Bill bill;
17   invariant bill.c>=0; // P_eH :  c ≥ 0
18   modifies bill.c;      // [c]
19 }
20 public class PaymentHandler { // Payment Processing Fee Handler
21   public void handleTotaling(TotalingEvent next){
22     next.bill().setC(next.bill().c()+1);   // c = c + 1
23   }
24   when TotalingEvent do handleTotaling;
25   public PaymentHandler(){ register(this); }
26 }
27 public class ShippingHandler { // Shipping Fee Handler
28   public void handleTotaling(TotalingEvent next){
29     next.bill().setC(next.bill().c()+0); //c = c + 0 NO FEE NOW
30   }
31   when TotalingEvent do handleTotaling;
32   public ShippingHandler(){ register(this); }
33 }
```

Figure 6.5: Full Delivery Example

To reason about the system requires to check that the handlers (lines 21-23 and 28-30) satisfy the event invariant and frame, and that the `Base.total()` method satisfies its specification.

i. The body of handler method `PaymentHandler.handleTotaling()` (line 22) should satisfy the event specification. Being

$S \equiv$
```
    next.bill().setC(next.bill().c()+1);   // c = c + 1
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c \geq 0,\ [c])$, as follows.

S

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule⟩

$(P_{[c+1/c,\ a/'a,\ c/'c]},\ P,\ [c])$

$\equiv$ ⟨by $P \equiv c \geq 0$⟩

$(\underline{c+1 \geq 0},\ c \geq 0,\ [c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ c \geq 0,\ [c])$ ∎

ii. The body of handler method `ShippingHandler.handleTotaling()` (line 29) should satisfy the event specification. Being

$S \equiv$
```
    next.bill().setC(next.bill().c()+0); //c = c + 0 NO FEE NOW
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c \geq 0,\ [c])$. That can be done as in (**i**.) .

iii. The body of method `Base.total()` (lines 11-12) should satisfy its specification. Being

$S \equiv$
```
    announce TotalingEvent(bill);
    bill.setA(bill.a()+bill.c()); // a = a + c
```

it is required to construct a proof for: $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$.

S

$\sqsupseteq$ ⟨by (ANNOUNCE) and (ASSIGNMENT) rules⟩

$(c \geq 0,\ c \geq 0,\ [c]);\ (\underline{P_{[a+c/a,\ a/'a,\ c/'c]},\ P,\ [a]})$

$\equiv$ ⟨by $P \equiv (a \geq a_0)$⟩

$(\underline{c \geq 0},\ c \geq 0,\ [c]);\ (a + c \geq a_0,\ a \geq a_0,\ [a])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(\underline{a \geq a_0} \wedge c \geq 0,\ c \geq 0,\ \underline{[c]});\ (a + c \geq a_0,\ a \geq a_0,\ [a])$

$\sqsupseteq$ $\langle$by frame$\rangle$

$(a \geq a_0 \wedge c \geq 0,\ \underline{a \geq a_0 \wedge c \geq 0},\ [c]);\ (a + c \geq a_0,\ a \geq a_0,\ [a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a \geq a_0 \wedge c \geq 0,\ a + c \geq a_0,\ [c]); (a + c \geq a_0,\ a \geq a_0,\ [a])$

$\sqsupseteq$ $\langle$by (SEQUENCE) rule$\rangle$

$(\underline{a \geq a_0 \wedge c \geq 0},\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$ ∎

As already mentioned in this approach the reasoning is constrained by the weakness or strength of the invariant the handlers satisfy.

6.2.2 II/EA Ptolemy Reasoning

Ptolemy follows the *Single-Delivery* approach. The runtime systems executes the first handler in the execution chain. It depends on whether this body contains *invoke* statements that the next handler will be executed or not. The announced code stands at the end of the execution chain. If there are handlers it depends on them if it will be executed. If there are no handlers it will be executed right away.

The **`event`** declaration includes the context variables (line 17), the specification that both the handlers and the announced-code must satisfy (lines 18,27,19) and the abstract algorithm (lines 20-26) the handlers should refine. To reason about the system requires to check that the handlers (lines 30-37, 42-49) and the announced code (line 12) satisfy the event specification and that the `Base.total()` method (lines 10-15) satisfies its specification (lines 7-9).

```
1  public class Base {
2    public void main(){
3      Bill bill=new Bill(100,8);
4      registerHandlers();
5      total(bill);
6    }
7    /*@ requires bill.c>=0; @*/                                   // c ≥ 0
8    /*@ modifies bill.a, bill.c; @*/                              // [a,c]
9    /*@ ensures bill.a()>=\old(bill.a())+\old(bill.c()); @*/ // a ≥ 'a + 'c
10   public void total(Bill bill){
11     announce TotalingEvent(bill) {
12      bill.setA(bill.a()+bill.c()); // a = a + c
13     }
14   }
15 }
16 public void event TotalingEvent { // (P_eH, Q_eH, ε_eH) ≡ (c ≥ 0, a ≥ 'a + 'c, [a,c])
17   Bill bill;
18   requires (bill.c()>=0);   // P_eH : c ≥ 0
19   modifies bill.a, bill.c; // [a,c]
20   assumes{
21     // spec. statement: (c ≥ 0, c ≥ 'c, [c])
22     requires (next.bill().c()>=0)
23       modifies next.bill().c
24       ensures (next.bill().c()>=old(next.bill().c()));
25     next.invoke(); // control flow: proceed with next handler
26   }
27   ensures (bill.a()>=old(bill.a())+old(bill.c())); //Q_eH : a ≥ 'a + 'c
28 }
29 public class PaymentHandler { // Payment Processing Fee Handler
30   public void handleTotaling(TotalingEvent next){
31     refining requires (next.bill().c()>=0)   // (c ≥ 0, c ≥ 'c, [c])
32              modifies next.bill().c
33              ensures (next.bill().c()>=old(next.bill().c())){
34       next.bill().setC(next.bill().c()+1); // c = c + 1
35     }
36     next.invoke();
37   }
38   when TotalingEvent do handleTotaling;
39   public PaymentHandler(){ register(this); }
40 }
```

Figure 6.6: Ptolemy Example

```
41 public class ShippingHandler { // Shipping Fee Handler
42   public void handleTotaling(TotalingEvent next)throws Throwable{
43     refining requires (next.bill().c()>=0)   // (c ≥ 0,  c ≥ 'c,  [c])
44              modifies next.bill().c
45              ensures (next.bill().c()>=old(next.bill().c())){
46       next.bill().setC(next.bill().c()+0); // c = c + 0 NO FEE NOW
47     }
48     next.invoke();
49   }
50   when TotalingEvent do handleTotaling;
51   public ShippingHandler(){ register(this); }
52 }
```

Figure 6.6: Ptolemy Example

For reasoning the handlers it is also required to reason about the **refining** statements in them.

i. The **refining** statement (lines 34-34) in `PaymentHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$

```
next.bill().setC(next.bill().c()+1); // c = c + 1
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c \geq {}'c,\ [c])$, as follows.

$$
\begin{aligned}
& S \\
\sqsupseteq \quad & \langle \text{by (ASSIGNMENT) rule} \rangle \\
& (P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]) \\
\equiv \quad & \langle \text{by } P \equiv c \geq {}'c \rangle \\
& (\underline{c+1 \geq c},\ c \geq {}'c,\ [c]); \\
\sqsupseteq \quad & \langle \text{by (CONSEQUENCE) rule} \rangle \\
& (c \geq 0,\ c \geq {}'c,\ [c]); \ \blacksquare
\end{aligned}
$$

ii. The body of handler method `PaymentHandler.handleTotaling()` (lines 31-36) must satisfy the event specification. Being

$S \equiv$

```
refining requires (next.bill().c()>=0)   // (c ≥ 0, c ≥ ′c, [c])
         modifies next.bill().c
         ensures (next.bill().c()>=old(next.bill().c())){
  next.bill().setC(next.bill().c()+1); // c = c + 1
}
next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0 + c_0,\ [a, c])$, as follows.

$$S$$

$\sqsupseteq$ $\langle$by results from (**i.**) and (INVOKE) rule$\rangle$

$$(c \geq 0,\ c \geq {}'c,\ [c]);\ (\underline{c \geq 0},\ a \geq {}'a + {}'c,\ [a, c])$$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$$(c \geq 0,\ c \geq {}'c,\ [c]);\ (\underline{c \geq 0 \wedge a + c \geq a_0 + c_0},\ a \geq {}'a + {}'c,\ [a, c])$$

$\sqsupseteq$ $\langle$by (PROPAGATION) rule$\rangle$

$$(c \geq 0,\ c \geq {}'c,\ [c]);$$

$$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ \underline{a \geq {}'a + {}'c \wedge {}'c \geq 0 \wedge {}'a + {}'c \geq a_0 + c_0},\ [a, c])$$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$$(c \geq 0,\ c \geq {}'c,\ [\underline{c}]);\ (c \geq 0 \wedge a + c \geq a_0 + c_0,\ a \geq a_0 + c_0,\ [a, c])$$

$\sqsupseteq$ $\langle$by (FRAME) rule$\rangle$

$$(\underline{c \geq 0},\ c \geq {}'c \wedge a = {}'a,\ [c]);\ (c \geq 0 \wedge a + c \geq a_0 + c_0,\ a \geq a_0 + c_0,\ [a, c])$$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$$(\underline{c \geq 0 \wedge a + c \geq a_0 + c_0},\ c \geq {}'c \wedge a = {}'a,\ [c]);$$

$$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ a \geq a_0 + c_0,\ [a, c])$$

$\sqsupseteq$ $\langle$by (PROPAGATION) rule$\rangle$

$$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ \underline{c \geq {}'c \wedge a = {}'a \wedge {}'c \geq 0 \wedge {}'a + {}'c \geq a_0 + c_0},\ [c]);$$

$$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ a \geq a_0 + c_0,\ [a, c])$$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ c \geq 0 \wedge a + c \geq a_0 + c_0,\ [c]);$

$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ a \geq a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (SEQUENCE) rule⟩

$(\underline{c \geq 0 \wedge a + c \geq a_0 + c_0},\ a \geq a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0 \wedge a = a_0 \wedge c = c_0,\ a \geq a_0 + c_0,\ [a, c])$ ∎

iii. The **refining** statement (lines 46-46) in `ShippingHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$
```
      next.bill().setC(next.bill().c()+0); // c = c + 0 NO FEE NOW
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c \geq {}'c,\ [c])$. That can be done as in (**i**).

iv. The body of handler method `ShippingHandler.handleTotaling()` (lines 43-38) must satisfy the event specification. Being

$S \equiv$
```
    refining requires (next.bill().c()>=0)   // (c ≥ 0,  c ≥ 'c,  [c])
             modifies next.bill().c
             ensures (next.bill().c()>=old(next.bill().c())){
      next.bill().setC(next.bill().c()+0); // c = c + 0 NO FEE NOW
    }
    next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0 + c_0,\ [a, c])$. The proof is similar to that in (**ii**).

v. The announced code (line 12) must satisfy the event specification. Beign

$S \equiv$
```
    bill.setA(bill.a()+bill.c()); // a = a + c
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a \geq {}'a + {}'c,\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule⟩

$(P_{[a+c/a,\ a/'a,\ c/'c]},\ P,\ [a])$

$\equiv \quad \langle$by $P \equiv a \geq 'a + 'c\rangle$

$(\underline{a + c \geq a + c},\ a \geq 'a + 'c,\ [a])$

$\sqsupseteq \quad \langle$by (CONSEQUENCE) rule$\rangle$

$(c \geq 0,\ a \geq 'a + 'c,\ [a, c])$ ∎

vi. The body of method `Base.total()` (lines 11-13) must satisfy its specification. Being

$S \equiv$
```
announce TotalingEvent(bill) {
 bill.setA(bill.a()+bill.c()); // a = a + c
}
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a \geq 'a + 'c,\ [a, c])$, as follows.

S

$\sqsupseteq \quad \langle$by (ANNOUNCE) rule and the results from (**v**.)$\rangle$

$(c \geq 0,\ a \geq 'a + 'c,\ [a, c])$ ∎

As pointed out by the author [55], Ptolemy's proof system is incomplete as it is incapable of modularly proving certain properties of valid programs.

6.2.3 II/EA PtolemyRely Reasoning

For more flexibility and completeness, PtolemyRely[55] extends Ptolemy [54, 7] by separating the handler's specification from the base-code specification (**`relies`** clause). Using this feature the handlers in the *Billing* example can be verified to strictly increase the final amount of a bill (*increasing* property) (lines 35,45) whilst the announced code just computes that amount (line 12).

```
 1 public class Base {
 2  public void main(){
 3   Bill bill=new Bill(100,8);
 4   registerHandlers();
 5   total(bill);
 6  }
 7  /*@ requires bill.c>=0; @*/                                    // c ≥ 0
 8  /*@ modifies bill.a, bill.c; @*/                               // [a,c]
 9  /*@ ensures bill.a()>=\old(bill.a())+\old(bill.c()); @*/ // a ≥ 'a + 'c
10  public void total(Bill bill){
11   announce TotalingEvent(bill) {
12    bill.setA(bill.a()+bill.c()); // a = a + c
13   }
14  }
15 }
16 public void event TotalingEvent { // (P_eH, Q_eH, ε_eH) ≡ (c ≥ 0,  a > 'a + 'c,  [a,c])
17                                   // (P_eB, Q_eB, ε_eB) ≡ (c ≥ 0,  a = 'a + 'c,  [a])
18  Bill bill;
19  relies requires bill.c()>=0                            // P_eB :  c ≥ 0
20         modifies bill.a                                 // ε_eB :  [a]
21         ensures bill.a()==old(bill.a())+old(bill.c()) // Q_eB :  a = 'a + 'c
22  requires (bill.c()>=0);   // P_eH :  c ≥ 0
23  modifies bill.a, bill.c; // ε_eH :  [a,c]
24  assumes{
25   requires next.bill().c()>=0 // (c ≥ 0,  c > 'c, [c])
26    modifies next.bill().c
27    ensures next.bill().c()>old(next.bill().c());
28   next.invoke(); // control flow: proceed with next handler
29  }
30  ensures (bill.a()>old(bill.a())+old(bill.c())); //Q_eH :  a > 'a + 'c
31 }
32 public class PaymentHandler { // Payment Processing Fee Handler
33  public void handleTotaling(TotalingEvent next){
34   refining requires next.bill().c()>=0 // (c ≥ 0,  c > 'c, [c])
35    modifies next.bill().c
36    ensures next.bill().c()>old(next.bill().c())
37     { next.bill().setC(next.bill().c()+1); } // c = c + 1
38   next.invoke();
39  }
40  when TotalingEvent do handleTotaling;
41  public PaymentHandler(){ register(this); }
42 }
```

Figure 6.7: PtolemyRely Example

```
43 public class ShippingHandler { // Shipping Fee Handler
44  public void handleTotaling(TotalingEvent next){
45   refining requires next.bill().c()>=0 //(c ≥ 0,  c > 'c, [c])
46    modifies next.bill().c
47    ensures next.bill().c()>old(next.bill().c())
48     { next.bill().setC(next.bill().c()+5); } // c = c + 5
49   next.invoke();
50  }
51  when TotalingEvent do handleTotaling;
52  public ShippingHandler(){ register(this); }
53 }
```

Figure 6.7: PtolemyRely Example

To reason about the system requires to check that the handlers (lines 33-39 and 44-50) satisfy the handlers specification, that the announced code (line 12) satisfy the *relies* specification and that the `Base.total()` method (lines 10-14) satisfies its specification (lines 7-9).

In the body of a handler, **`refining`** statements are reasoned about using their own specifications and **`invoke`** statements are reasoned about using the non-deterministic choice between the handlers specification and the *relies* specification.

i. The **`refining`** statement (lines 37-37) in `PaymentHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$
```
{ next.bill().setC(next.bill().c()+1); } // c = c + 1
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c > {}'c,\ [c])$, as follows.

$$
\begin{array}{ll}
 & S \\
\sqsupseteq & \langle \text{by (ASSIGNMENT) rule} \rangle \\
 & (P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]) \\
\equiv & \langle \text{by } P \equiv c > {}'c \rangle \\
 & (\underline{c+1 > c},\ c > {}'c,\ [c]);
\end{array}
$$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ c > {'c},\ [c])$; ∎

ii. The body of handler method `PaymentHandler.handleTotaling()` (lines 34-38) must satisfy the event specification. Being

$S \equiv$

```
refining requires next.bill().c()>=0 // (c ≥ 0,  c > 'c, [c])
 modifies next.bill().c
 ensures next.bill().c()>old(next.bill().c())
  { next.bill().setC(next.bill().c()+1); } // c = c + 1
next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a > a_0 + c_0,\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by results from (**i**.) and (INVOKE) rule and $((c \geq 0, a > {'a} + {'c}, [a, c]) \Box (c \geq 0, a = {'a} + {'c}, [a])) \equiv (c \geq 0, a \geq {'a} + {'c}, [a, c])$⟩

$(c \geq 0,\ c > {'c},\ [c]);\ (\underline{c \geq 0},\ a \geq {'a} + {'c},\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ c > {'c},\ [c]);\ (\underline{c \geq 0 \wedge a + c > a_0 + c_0},\ a \geq {'a} + {'c},\ [a, c])$

$\sqsupseteq$ ⟨by (PROPAGATION) rule⟩

$(c \geq 0,\ c > {'c},\ [c]);$

$(c \geq 0 \wedge a + c > a_0 + c_0,\ \underline{a \geq {'a} + {'c} \wedge {'c} \geq 0 \wedge {'a} + {'c} > a_0 + c_0},\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ c > {'c},\ [\underline{c}]);\ (c \geq 0 \wedge a + c > a_0 + c_0,\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (FRAME) rule⟩

$(\underline{c \geq 0},\ c > {'c} \wedge a = {'a},\ [c]);\ (c \geq 0 \wedge a + c > a_0 + c_0,\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(\underline{c \geq 0 \wedge a + c \geq a_0 + c_0},\ c > {'c} \wedge a = {'a},\ [c]);$

$(c \geq 0 \wedge a + c > a_0 + c_0,\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (PROPAGATION) rule⟩

$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ \underline{c > {'c} \wedge a = {'a} \wedge {'c} \geq 0 \wedge {'a} + {'c} \geq a_0 + c_0},\ [c]);$

$(c \geq 0 \wedge a + c > a_0 + c_0,\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0 \wedge a + c \geq a_0 + c_0,\ c \geq 0 \wedge a + c > a_0 + c_0,\ [c]);$

$(c \geq 0 \wedge a + c > a_0 + c_0,\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (SEQUENCE) rule⟩

$(\underline{c \geq 0 \wedge a + c \geq a_0 + c_0},\ a > a_0 + c_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0 \wedge a = a_0 \wedge c = c_0,\ a > a_0 + c_0,\ [a, c])$ ∎

iii. The **`refining`** statement (lines 48-48) in `ShippingHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$
```
    { next.bill().setC(next.bill().c()+5); } // c = c + 5
```

it is required to prove that $S \sqsupseteq (c \geq 0,\ c > {'c},\ [c])$. That can be done as in (**i**).

iv. The body of handler method `ShippingHandler.handleTotaling()` (lines 45-49) must satisfy the event specification. Being

$S \equiv$
```
  refining requires next.bill().c()>=0 //(c ≥ 0,  c > 'c, [c])
   modifies next.bill().c
   ensures next.bill().c()>old(next.bill().c())
    { next.bill().setC(next.bill().c()+5); } // c = c + 5
  next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a > a_0 + c_0,\ [a, c])$. The proof is similar to that in (**ii**).

v. The announced code (line 12) must satisfy the event *relies* specification. Being

$S \equiv$
```
   bill.setA(bill.a()+bill.c()); // a = a + c
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a = {'a} + {'c},\ [a])$, as follows.

S

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule⟩

$(P_{[a+c/a,\ a/'a,\ c/'c]},\ P,\ [a])$

$\equiv$ ⟨by $P \equiv a = {'a} + {'c}$⟩

$(\underline{a + c = a + c},\ a = {'a} + {'c},\ [a])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ a = {'a} + {'c},\ [a])$ ∎

vi. The body of method `Base.total()` (lines 11-13) must satisfy its specification. Being

$S \equiv$

```
announce TotalingEvent(bill) {
 bill.setA(bill.a()+bill.c()); // a=a+c
}
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a \geq {'a} + {'c},\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by (ANNOUNCE) rule and considering the results from (**v**.)⟩

$(c \geq 0,\ a > {'a} + {'c},\ [a, c]) \square (c \geq 0,\ a = {'a} + {'c},\ [a])$

$\equiv$ ⟨by definition of $\square$, predicate calculus and set theory⟩

$(c \geq 0,\ a \geq {'a} + {'c},\ [a, c])$ ∎

By separating the specifications for the handlers and the announced-code, PtolemyRely proof system is capable of enforcing the *increasing* property and still verify the *Billing* system.

6.2.4 Modular Behavior-Preserving Reasoning

As detailed in section 5.4.2.3, if the specification for the handlers refines the least upper bound of the specifications for all blocks of announced-code then every announce **statement** will refine the original announced code. Example in Figure 6.8 illustrates that.

```
1  public class Base {
2   public void main(){
3    Bill bill=new Bill(100,8);
4    registerHandlers();
5    total(bill);
6   }
7   /*@ requires bill.c>=0; @*/                                  // c ≥ 0
8   /*@ modifies bill.a, bill.c; @*/                             // [a, c]
9   /*@ ensures bill.a()>=\old(bill.a())+\old(bill.c()); @*/ // a ≥ 'a
10  public void total(Bill bill){
11   announce TotalingEvent(bill) {
12    bill.setA(bill.a()+bill.c()); // a = a + c :  (c ≥ 0,  a ≥ 'a, [c, a])
13   }
14   announce TotalingEvent(bill) {
15    bill.setA(bill.a()+1); // a = a + 1 :  (tt,  a ≥ 'a, [c, a])
16   }
17  }
18 }
19 public void event TotalingEvent { // (P_eH, Q_eH, ε_eH) ≡ (tt,  a ≥ 'a,  [c, a])
20  Bill bill;
21  relies requires next.bill().c()>=0       // P_eB :  c ≥ 0
22         modifies bill.a, bill.c           // ε_eB :  [c, a]
23         ensures bill.a()>=old(bill.a())) // Q_eB :  a ≥ 'a
24  requires true;   // P_eH :  tt
25  modifies bill.c, bill.a; // ε_eH :  [c, a]
26  assumes{
27  requires true //(tt,  c ≥ 0,  [c])
28    modifies next.bill().c
29    ensures next.bill().c()>=0;
30   next.invoke(); // control flow: proceed with next handler
31  }
32  ensures (bill.a()>old(bill.a())+old(bill.c())); //Q_eH :  a ≥ 'a
33 }
```

Figure 6.8: Modular Behavior-Preserving Example

```
34 public class PaymentHandler { // Payment Processing Fee Handler
35  public void handleTotaling(TotalingEvent next){
36   refining requires true //(tt,  c ≥ 0,  [c])
37    modifies next.bill().c
38    ensures next.bill().c()>=0
39     { if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
40       next.bill().setC(next.bill().c()+1); }      // c = c + 1
41   next.invoke();
42  }
43  when TotalingEvent do handleTotaling;
44  public PaymentHandler(){ register(this); }
45 }
46 public class ShippingHandler { // Shipping Fee Handler
47  public void handleTotaling(TotalingEvent next)throws Throwable{
48   refining requires true //(tt,  c ≥ 0,  [c])
49    modifies next.bill().c
50    ensures next.bill().c()>=0
51     { if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
52       next.bill().setC(next.bill().c()+5); }      // c = c + 5
53   next.invoke();
54  }
55  when TotalingEvent do handleTotaling;
56  public ShippingHandler(){ register(this); }
57 }
```

Figure 6.8: Modular Behavior-Preserving Example

The `Base.total()` method (lines 10-19) has two event announcements of event `TotalingEvennt`. One can compute a handlers specification that refines each block of announced code:

$(c \geq 0,\ a \geq {}'a,\ [a, c]) \sqcup (tt,\ a \geq {}'a,\ [a, c])$

$\equiv$ ⟨by Definition of $\sqcup$⟩

$(c \geq 0 \vee tt,\ ({}'c \geq 0 \Rightarrow a \geq {}'a) \wedge (tt \Rightarrow a \geq {}'a),\ [a, c])$

$\equiv$ ⟨by predicate calculus⟩

$(tt,\ (c \geq 0 \Rightarrow a \geq {}'a) \wedge (a \geq {}'a),\ [a, c])$

$\equiv$ ⟨by predicate calculus⟩

$(tt,\ a \geq {}'a,\ [a])$

$\Longrightarrow$ ⟨by convention⟩

$$(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \equiv (tt,\ a \geq {}'a,\ [a])$$

And a base-code specification that is refined by each such behavior:

$$(c \geq 0,\ a \geq {}'a,\ [a,c]) \sqcap (tt, a \geq {}'a,\ [a,c])$$

$\equiv$ ⟨by Definition of $\sqcap$⟩

$$(c \geq 0 \wedge tt,\ a \geq {}'a \vee a \geq {}'a,\ [a,c])$$

$\equiv$ ⟨by predicate calculus⟩

$$(c \geq 0,\ a \geq {}'a,\ [a,c])$$

$\Longrightarrow$ ⟨by convention⟩

$$(P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \equiv (c \geq 0,\ a \geq {}'a,\ [a,c])$$

To reason about the system requires to check that the handlers (lines 35-42 and 47-54) satisfy the handlers specification, that the announced code (line 12 and 15) satisfy the *relies* specification and that the `Base.total()` method (lines 10-17) satisfies its specification (lines 7-9). In the body of a handler, **`refining`** statements are reasoned about using their own specifications and **`invoke`** statements are reasoned about using the non-deterministic choice between the handlers specification and the *relies* specification.

i. The **`refining`** statement (lines 39-40) in `PaymentHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$

```
{ if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
  next.bill().setC(next.bill().c()+1); }      // c = c + 1
```

it is required to prove that $S \sqsupseteq (tt,\ c \geq 0, [c])$, as follows.

$$S$$

$$\sqsupseteq \quad \left\langle \begin{array}{l}\text{by (CONDITIONAL) and (ASSIGNMENT) rules and } \{c < 0 \wedge tt\}\ c = 0\ \{c \geq 0\} \\ \text{and } \{\neg(c < 0) \wedge tt\}\ skip\ \{c \geq 0\}\end{array} \right\rangle$$

$$(tt,\ c \geq 0,\ [c]);\ \underline{(P[c+1/c,\ a/'a,\ c/'c],\ P,\ [c])}$$

$$\equiv \quad \langle \text{by } P \equiv c \geq 0 \rangle$$

$$(tt,\ \underline{c \geq 0},\ [c]);\ (c+1 \geq 0,\ c \geq 0,\ [c])$$

$$\sqsupseteq \quad \langle \text{by (CONSEQUENCE) rule} \rangle$$

$$(tt,\ c+1 \geq 0,\ [c]);\ (c+1 \geq 0,\ c \geq 0,\ [c])$$

$$\sqsupseteq \quad \langle \text{by (SEQUENCE) rule} \rangle$$

$$(tt,\ c \geq 0,\ [c])\ \blacksquare$$

ii. The body of handler method `PaymentHandler.handleTotaling()` (lines 36-41) must satisfy the event handlers specification. Being

$S \equiv$

```
refining requires true //(tt,  c ≥ 0,  [c])
 modifies next.bill().c
 ensures next.bill().c()>=0
  { if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
    next.bill().setC(next.bill().c()+1); }      // c = c + 1
next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0,\ a \geq a_0,\ [c, a])$, as follows.

$$S$$

$$\sqsupseteq \quad \left\langle \begin{array}{l}\text{by results from (\textbf{i}.) and (INVOKE) rule that is reasoned about as} \\ (P_{eB}, Q_{eB}, \varepsilon_{eB}) \equiv (c \geq 0,\ a \geq\ 'a,\ [c, a])\end{array} \right\rangle$$

$$(tt,\ c \geq 0,\ [c]);\ (\underline{c \geq 0},\ a \geq\ 'a,\ [c, a])$$

$$\sqsupseteq \quad \langle \text{by (CONSEQUENCE) rule} \rangle$$

$$(tt,\ c \geq 0,\ [c]);\ (\underline{c \geq 0 \wedge a \geq a_0},\ a \geq\ 'a,\ [c, a])$$

$$\sqsupseteq \quad \langle \text{by (PROPAGATION) rule} \rangle$$

$(tt,\ c \geq 0,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ \underline{a \geq {}'a \wedge {}'c \geq 0 \wedge {}'a \geq a_0},\ [c,a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(tt,\ c \geq 0,\ \underline{[c]});\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (FRAME) rule$\rangle$

$(\underline{tt},\ c \geq 0 \wedge a = {}'a,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(\underline{a \geq a_0},\ c \geq 0 \wedge a = {}'a,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (PROPAGATION) rule$\rangle$

$(a \geq a_0,\ \underline{c \geq 0 \wedge a = {}'a \wedge {}'a \geq a_0},\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a \geq a_0,\ c \geq 0 \wedge a \geq a_0,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (SEQUENCE) rule$\rangle$

$(a \geq a_0,\ a \geq a_0,\ [c,a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a = a_0 \wedge c = c_0,\ a \geq a_0,\ [c,a])$ ∎

iii. The **`refining`** statement (lines 51-52) in `ShippingHandler.handleTotaling()` must satisfy its specification. Being

$S \equiv$

```
{ if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
  next.bill().setC(next.bill().c()+5); }     // c = c + 5
```

it is required to prove that $S \sqsupseteq (tt,\ c \geq 0, [c])$. That can be done as in (**i**).

iv. The body of handler method `ShippingHandler.handleTotaling()` (lines 48-53) must satisfy the event specification. Being

$S \equiv$

```
refining requires true //(tt,  c ≥ 0,  [c])
 modifies next.bill().c
 ensures next.bill().c()>=0
  { if(next.bill().c()<0) next.bill().setC(0); // c < 0 ⇒ c = 0
    next.bill().setC(next.bill().c()+5); }     // c = c + 5
next.invoke();
```

it is required to prove that $S \sqsupseteq (a = a_0 \wedge c = c_0,\ a \geq a_0,\ [c, a])$. The proof is similar to (**ii**).

v. The announced code (line 12) must satisfy the event *relies* specification. Being

$S \equiv$

```
bill.setA(bill.a()+bill.c()); // a = a + c :  (c ≥ 0,  a ≥ 'a, [c, a])
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a \geq {}'a,\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule⟩

$(P_{[a+c/a,\ a/{}'a,\ c/{}'c]},\ P,\ [a])$

$\equiv$ ⟨by $P \equiv a \geq {}'a$⟩

$(\underline{a + c \geq a},\ a \geq {}'a,\ [a])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ a \geq {}'a,\ [a, c])$ ∎

vi. The announced code (line 15) must satisfy the event *relies* specification. Being

$S \equiv$

```
bill.setA(bill.a()+1); // a = a + 1 :  (tt,  a ≥ 'a, [c, a])
```

It is required to construct a proof for: $S \sqsupseteq (c \geq 0,\ a \geq {}'a,\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule⟩

$(P_{[a+1/a,\ a/{}'a,\ c/{}'c]},\ P,\ [a])$

$\equiv$ ⟨by $P \equiv a \geq {}'a$⟩

$(\underline{a + 1 \geq a},\ a \geq {}'a,\ [a])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(c \geq 0,\ a \geq {'a},\ [a, c])$ ∎

vii. The body of method `Base.total()` (lines 11-16) must satisfy its specification. Being

$S \equiv$
```
announce TotalingEvent(bill) {
 bill.setA(bill.a()+bill.c()); // a = a + c :  (c ≥ 0,  a ≥ 'a, [c, a])
}
announce TotalingEvent(bill) {
 bill.setA(bill.a()+1); // a = a + 1 :  (tt,  a ≥ 'a, [c, a])
}
```

It is required to construct a proof for: $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$:

S

$\sqsupseteq$ $\langle$by results **v**. and **vi**. and (ANNOUNCE) rule$\rangle$

$(c \geq 0,\ a \geq {'a},\ [a, c]);\ (\underline{tt},\ a \geq {'a},\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(c \geq 0,\ a \geq {'a},\ [a, c]);\ (\underline{a \geq a_0},\ a \geq {'a},\ [a, c])$

$\sqsupseteq$ $\langle$by (PROPAGATION) rule$\rangle$

$(c \geq 0,\ a \geq {'a},\ [a, c]);\ (a \geq a_0,\ \underline{a \geq {'a} \wedge {'a} \geq a_0},\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(\underline{c \geq 0},\ a \geq {'a},\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(\underline{c \geq 0 \wedge a \geq a_0},\ a \geq {'a},\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (PROPAGATION) rule$\rangle$

$(c \geq 0 \wedge a \geq a_0,\ \underline{a \geq {'a} \wedge {'c} \geq 0 \wedge {'a} \geq a_0},\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (SEQUENCE) rule$\rangle$

$(\underline{c \geq 0 \wedge a \geq a_0},\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$ ∎

This approach preserves the behavior of the original (without events) base code, which guarantees that the system behaves as expected, but for that it imposes a strong specification on the handlers, the least upper bound of all the blocks of announced code.

6.2.5 Non-Modular Individual Refinement Reasoning

If at each event announcement every applicable handler proceed-composed with the current announced-code specification satisfies this same specification then the **announce** statement preserves that specification. In this approach the event has no specification and instead each handler is checked to satisfy the previous condition at each event announcement.

```
1  public class Base {
2   public void main(){
3    Bill bill=new Bill(100,8);
4    registerHandlers();
5    total(bill);
6   }
7   /*@ requires bill.c>=0; @*/                                  // c ≥ 0
8   /*@ modifies bill.a, bill.c; @*/                             // [a,c]
9   /*@ ensures bill.a()>=\old(bill.a())+\old(bill.c()); @*/ // a ≥ 'a
10  public void total(Bill bill){
11   announce TotalingEvent(bill) {
12    bill.setA(bill.a()+bill.c()); // a = a + c :  (c ≥ 0,  a ≥ 'a, [a,c])
13   }
14   announce TotalingEvent(bill) {
15    bill.setA(bill.a()+1); // a = a + 1 :  (tt,  a ≥ 'a, [a,c])
16   }
17  }
18 }
```

Figure 6.9: Non-Modular Individual Refinement Example

```
19 public void event TotalingEvent {
20  Bill bill;
21 }
22 public class PaymentHandler { // Payment Processing Fee Handler
23  public void handleTotaling(TotalingEvent next){
24   next.bill().setC(next.bill().c()+1); // c = c + 1
25   next.invoke();
26  }
27  when TotalingEvent do handleTotaling;
28  public PaymentHandler(){ register(this); }
29 }
30 public class ShippingHandler { // Shipping Fee Handler
31  public void handleTotaling(TotalingEvent next){
32   next.bill().setC(next.bill().c()+5); // c = c + 5
33   next.invoke();
34  }
35  when TotalingEvent do handleTotaling;
36  public ShippingHandler(){ register(this); }
37 }
```

Figure 6.9: Non-Modular Individual Refinement Example

To reason about the system requires to check that each handler (lines 23-26 and 31-34) proceed-composed with the specification at each announcement (line 12 and 15) satisfy that specification.

i. The body of handler method `PaymentHandler.handleTotaling()` (lines 24-25) proceed composed with the specification for the first announcement (line 12) should satisfy that specification. Being

$S \equiv$

```
next.bill().setC(next.bill().c()+1); // c = c + 1
next.invoke();
```

it is required to prove that $(S \circledcirc (c \geq 0,\ a \geq {}'a, [a, c])) \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$, as follows.

$S \circledcirc (c \geq 0,\ a \geq {}'a, [a, c])$

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule and definition of $\circledcirc$⟩

$(P_{[c+1/c,\ a/'a,\ c/'c]},\ P,\ [c]);\ (\underline{c \geq 0},\ a \geq 'a,\ [a,c])$

$\sqsupseteq \quad \langle$by (CONSEQUENCE) rule$\rangle$

$(P_{[c+1/c,\ a/'a,\ c/'c]},\ P,\ [c]);\ (\underline{c \geq 0 \wedge a \geq a_0},\ a \geq 'a,\ [a,c])$

$\sqsupseteq \quad \langle$by (PROPAGATION) rule $\rangle$

$(P_{[c+1/c,\ a/'a,\ c/'c]},\ P,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ \underline{a \geq 'a \wedge 'c \geq 0 \wedge 'a \geq a_0},\ [a,c])$

$\sqsupseteq \quad \langle$by (CONSEQUENCE) rule $\rangle$

$(\underline{P_{[c+1/c,\ a/'a,\ c/'c]},\ P,\ [c]});\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [a,c])$

$\equiv \quad \langle$by $P \equiv c \geq 0 \wedge a \geq a_0\rangle$

$(\underline{c+1 \geq 0 \wedge a \geq a_0},\ c \geq 0 \wedge a \geq a_0,\ [c]);\ (c \geq 0 \wedge a \geq a_0,\ a \geq a_0,\ [a,c])$

$\equiv \quad \langle$by (SEQUENCE) rule$\rangle$

$(\underline{c+1 \geq 0 \wedge a \geq a_0},\ a \geq a_0,\ [a,c])$

$\equiv \quad \langle$by (CONSEQUENCE) rule$\rangle$

$(a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a,c])$ ∎

ii. The body of handler method `ShippingHandler.handleTotaling()` (lines 32-33) proceed composed with the specification for the first announcement (line 12) should satisfy that specification. Being

$S \equiv$
```
next.bill().setC(next.bill().c()+5); // c = c + 5
next.invoke();
```

it is required to prove that $(S \circledcirc (c \geq 0,\ a \geq\ 'a, [a,c])) \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a,c])$. That can be done as in (**i**.)

iii. The body of handler method `PaymentHandler.handleTotaling()` (lines 24-25) proceed composed with the specification for the second announcement (line 15) should satisfy that specification. Being

$S \equiv$
```
next.bill().setC(next.bill().c()+1); // c = c + 1
next.invoke();
```

it is required to prove that $(S \circledcirc (tt,\ a \geq {}'a,\ [a,c])) \sqsupseteq (a = a_0 \wedge c = c_0,\ a \geq a_0,\ [a,c])$, as follows.

$$S \circledcirc (tt,\ a \geq {}'a,\ [a,c])$$

$\sqsupseteq$ ⟨by (ASSIGNMENT) rule and definition of $\circledcirc$⟩

$$(P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]);\ (\underline{tt},\ a \geq {}'a,\ [a,c])$$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$$(P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]);\ (\underline{a \geq a_0},\ a \geq {}'a,\ [a,c])$$

$\sqsupseteq$ ⟨by (PROPAGATION) rule ⟩

$$(P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]);\ (a \geq a_0,\ \underline{a \geq {}'a \wedge {}'a \geq a_0},\ [a,c])$$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule ⟩

$$(P_{[c+1/c,\ a/{}'a,\ c/{}'c]},\ P,\ [c]);\ (a \geq a_0,\ a \geq a_0,\ [a,c])$$

$\equiv$ ⟨by $P \equiv a \geq a_0$⟩

$$(a \geq a_0,\ a \geq a_0,\ [c]);\ (a \geq a_0,\ a \geq a_0,\ [a,c])$$

$\equiv$ ⟨by (SEQUENCE) rule⟩

$$(\underline{a \geq a_0},\ a \geq a_0,\ [a,c])$$

$\equiv$ ⟨by (CONSEQUENCE) rule⟩

$$(a = a_0 \wedge c = c_0,\ a \geq a_0,\ [a,c])\ \blacksquare$$

iv. The body of handler method `ShippingHandler.handleTotaling()` (lines 32-33) proceed composed with the specification for the second announcement (line 15) should satisfy that specification. Being

$S \equiv$
```
next.bill().setC(next.bill().c()+5); // c = c + 5
next.invoke();
```

it is required to prove that $(S \circledcirc (tt,\ a \geq {}'a,\ [a,c])) \sqsupseteq (a = a_0 \wedge c = c_0,\ a \geq a_0,\ [a,c])$. It can be done as in (**iii**.)

v. The body of method `Base.total()` (lines 11-17) must satisfy its specification. Being

$S \equiv$
```
 announce TotalingEvent(bill) {
  bill.setA(bill.a()+bill.c()); // a = a + c :  (c ≥ 0,  a ≥ 'a, [c, a])
 }
 announce TotalingEvent(bill) {
  bill.setA(bill.a()+1); // a = a + 1 :  (tt,  a ≥ 'a, [c, a])
 }
}
```

It is required to construct a proof for: $S \sqsupseteq (a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$, as follows.

S

$\sqsupseteq$ ⟨by the results from (**i**.,**ii**.,**iii**.,**iv**.) and the (ANNOUNCE) rule that keeps the announced code behavior.⟩

$(c \geq 0,\ a \geq {}'a,\ [a, c]);\ (\underline{tt},\ a \geq {}'a,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(c \geq 0,\ a \geq {}'a,\ [a, c]);\ (\underline{a \geq a_0},\ a \geq {}'a,\ [a, c])$

$\sqsupseteq$ ⟨by (PROPAGATION) rule⟩

$(c \geq 0,\ a \geq {}'a,\ [a, c]);\ (a \geq a_0,\ \underline{a \geq {}'a \wedge {}'a \geq a_0},\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(\underline{c \geq 0},\ a \geq {}'a,\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(\underline{a \geq a_0 \wedge c \geq 0},\ a \geq {}'a,\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ ⟨by (PROPAGATION) rule⟩

$(a \geq a_0 \wedge c \geq 0,\ \underline{a \geq {}'a \wedge {}'a \geq a_0 \wedge {}'c \geq 0},\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ ⟨by (CONSEQUENCE) rule⟩

$(a \geq a_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c]);\ (a \geq a_0,\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ ⟨by (SEQUENCE) rule⟩

$(\underline{a \geq a_0 \wedge c \geq 0},\ a \geq a_0,\ [a, c])$

$\sqsupseteq$ $\langle$by (CONSEQUENCE) rule$\rangle$

$(a = a_0 \wedge c = c_0 \wedge c \geq 0,\ a \geq a_0,\ [a, c])$ ∎

This approach gives for flexibility to the handlers as they are reasoned about considering the concrete specification of each announced code, instead of a weaker generalization of them. The drawback of the approach is that it is not modular.

CHAPTER 7: CONCLUSION

The aimed problem of this dissertation has been to do formal reasoning for specific scenarios in implicit-invocation and aspect-oriented like languages. The scenarios correspond to different configurations of important tradeoffs faced when reasoning programs developed with these types of languages.

Several reasoning tradeoffs have been identified. Scenarios derived from making choices regarding these tradeoffs has been configured and characterized. Proof rules for reasoning about programs in the different scenarios were formalized. Operational semantics for the new or updated languages constructs was defined. The soundness of the proof rules, with regard to the defined semantics, was demonstrated. Reasoning examples illustrating the use of the proof rules were also developed.

7.1 Tradeoffs and Reasoning

The main identified tradeoffs and how they affect the reasoning process is summarized in the following:

Modular vs. Non-Modular: In modular reasoning, component invocations are reasoned about using the components' specifications instead of their implementations. This approach has may advantages but may be limiting in some situations. If at a certain point many components may be invoked, like in OO dynamic dispatch or II event announcement, then a common specification that abstracts them all would have to be used, maybe loosing useful reasoning information. In this cases the modular-vs-non-modular tradeoff is important. The Full Delivery (5.4.1), PtolemyRely (5.4.2.1), Ptolemy (5.4.2.2) and Behavior-Preserving Modular (5.4.2.3) scenarios use modular reasoning while the Non-Modular Individual Re-

finement Scenario (5.4.2.4) uses non-modular reasoning.

Case analysis vs. Abstraction: This tradeoff is closely related to the previous one. In non-modular reasoning, to reason about a client component one needs to consider all the components that could be invoked at a certain point, so a case by case analysis is required. In modular reasoning all the invoked components are abstracted by a common specification, an it is used to reason about the invocation at that given point.

Full vs. Single Delivery: In event based implicit invocation systems many handlers could be registered for an event. Whether the system invokes them all, like in C# *delegates* [22, 46] and Java *JavaBeans* [61] or just one, like PtolemyRely [55]) and Ptolemy [54], is a tradeoff considered in configuring the reasoning scenarios. In Full-Delivery each handler must leave the system in an state such that the next handler can be invoked, so an invariant specification should be imposed in all of them, and this invariant can be use to reason about the event announcement. In Single-Delivery a specification is defined for all the handlers and this specification can be used to reason both the event announcement itself and any next-handler invocation in the body of a handler.

Explicit vs. Implicit Invocation: Explicit invocation, as supported by OO languages, has the advantage that the specification of the unique invoked component can be used to reason about the invocation; but it has the disadvantage that tightly couples the invoking and invoked components. Implicit invocation, as in event based and AO languages, does not suffer of the coupling problem, but makes it harder to reason about the event announcement: non-modular reasoning can be used or a common specification can be imposed on all handlers, restoring the modular reasoning.

Explicit vs. Implicit Announcement: With Explicit Announcement, the place where an event occurs in the client code is explicit, that being done by some kind of *announce* construct.

Then, the announcement can be reasoned about using explicit or implicit invocation. On the other hand, in Implicit Announcement, as in AO languages, the client remains oblivious of where and what functionality may be invoked. One partial solution proposed [35] to reason about system with implicit announcement is to do it in two phases. A whole program non-modular analysis first determine the advised points and then a modular reasoning is applied to each one of them, using an implicit invocation approach.

7.2 Scenarios

Different scenarios were derived by taking choices regarding the identified tradeoffs. They are summarized as follows:

II/EA Full Delivery Scenario: This scenario is configured by using explicit event announcement, implicit invocation of all registered handlers and modular reasoning. The invariant specification is included in the event declaration. Handlers are modularly reasoned about against this specification and event announcements are reasoned about using this invariant.

II/EA Ptolemy Modular Scenario: This scenario correspond to the original definition of Ptolemy language [54] and its specification and verification features [7]. It follows the single-delivery strategy: at event announcement only one registered handler is invoked. It is left under the developer control to include *invoke* statements in the body of a handler for invoking the next handler (or announced code). This scenario has all the advantages of explicit announcement, implicit invocation, single delivery and modular reasoning. *Announce* and *invoke* statements are reasoned about against the event specification, included in the event declaration. Handlers and announced-code must be reasoned about against that event specification. One weakness of imposing the same specification on both the handlers and announced code is

that this does not allow to enforce certain properties that depend on having different behaviors for them.

II/EA PtolemyRely Modular Scenario: This scenario correspond to an enhancement made by the author to the Ptolemy specification and verification features [55, 56], for separating the specifications for handlers and announced code. It is also an explicit announcement, implicit invocation, single delivery and modular reasoning scenario. Announce statements are reasoned about as the non-deterministic choice between the handlers specification and the particular announced-code behavior. *Invoke* statements are reasoned about as the non-deterministic choice between the handlers specification and the announced-code specification, as stated in the event declaration. Handlers must be reasoned about against that handlers specification and announced-code against announced-code specification. This scenario is flexible and more complete than the Ptolemy scenario, as any Ptolemy system can be encoded as a PtolemyRely system using the same specification for handlers and announced code.

II/EA Behavior-Preserving Modular Scenario: This scenario is not based on extra language features but in a developer discipline used on top of PtolemyRely language. The starting point is the specifications of the blocks of announced code. The discipline consist in setting the event announced-code specification as the greatest lower bound of these blocks' specifications and the event handlers specification as the least upper bound of the blocks' specifications. As demonstrated in section 5.4.2.3, using these settings *announce* statements preserve the corresponding blocks specifications, keeping the original client behavior. This scenario has the advantage of preserving the original behavior but it could impose a strong specification on the handlers, leaving to their implementers the responsibility of coping with this.

II/EA Behavior-Preserving Non-Modular Scenario: This scenario explores the non-modular ap-

proach while preserving the original client behavior. The event declaration has no specifications at all. At each event announcement every applicable handler, composed with the current announced-block specification is reasoned about against this same specification. As demonstrated in sections 5.4.2.4 and 6.1.3, *announce* statements preserve the corresponding blocks specifications, keeping the original client behavior. This approach gives for flexibility to the handlers as they are reasoned about considering the concrete specification of each announced block, instead of a weaker generalization of them. The drawback of the approach is that it is not modular.

AO Scenario: The AO scenario is similar to the Implicit Invocation scenarios. The main difference is that the events are considered to be triggered not explicitly by `announce` statements but implicitly by the execution of the shadows picked by pointcuts. The pointcut declaration in AO is analogous to the event declaration in II. The shadows corresponds to the blocks of announced code and the pieces of advice correspond to the handlers. Pointcut declarations can be annotated with specifications similar to those used in II events (like in Ptolemy or PtolemyRely). Reasoning about AO scenarios can be done in a two-phase approach. First, for each join point in the base code all matching pointcuts are identified, and all the pieces of advice for those pointcuts are considered the handlers for the join point event. Then the reasoning strategies used for the single delivery II/EA scenario can be used. This scenario provides all the flexibility of AO but is not modular.

7.3 Future Work

Some lines of investigation are suggested for further work.

The starting point of many of the scenarios is the specifications for the blocks of code (announced code) triggering events. Currently this specifications are supposed to be provided by the developer.

An interesting venue of research is the automatic inference of these specifications. A initial approach is to use weakest precondition computations starting from the postcondition of the method containing the blocks, but this assumes that this method has a specification itself. More work is required in this area.

A detailed design of the specifications features that would be required for annotating AO pointcut declarations, similar to those in PtolemyRely, is also an area for future work.

An entire Hoare-style logic, including rules like the ones in section 6.2, for reasoning in the presence of two-state postcondition would be very helpful for doing that in a more concise and intuitive way.

The Behavior-Preserving Modular scenario presented in section 5.4.2.3 depends on a developer discipline consisting in setting the event announced-code specification as the greatest lower bound of these blocks' specifications and the event handlers specification as the least upper bound of the blocks' specifications. Exploring the possibility of enforcing this automatically is also a topic that can be developed.

LIST OF REFERENCES

[1] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, NY, 1997. Expanded in DEC SRC report 161.

[2] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, Nederlandse Philips Bedrijven B. V., January 1989. Superseded by a later version in April 1989.

[3] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998.

[4] Mehdi Bagherzadeh, Robert Dyer, Rex D. Fernando, Hridesh Rajan, and Jose Sanchez. Modular reasoning in the presence of event subtyping. Technical Report 14-02b, Iowa State University, Dept. of Computer Sc., 2014.

[5] Mehdi Bagherzadeh, Gary T. Leavens, and Robert Dyer. Applying translucid contracts for modular reasoning about aspect and object oriented events. In *Proceedings of the 10th international workshop on Foundations of aspect-oriented languages*, FOAL '11, pages 31–35, New York, NY, USA, 2011. ACM.

[6] Mehdi Bagherzadeh, Hridesh Rajan, and Ali Darvish. On exceptions, events and observer chains. Technical Report 12-12, Iowa State University, Dept. of Computer Sc., 2012.

[7] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In

Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11, pages 141–152, New York, NY, USA, 2011. ACM.

[8] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Local reasoning for global invariants, part i: Region logic. *Journal of the ACM*, 60(3):18:1–18:56, June 2013.

[9] Mike Barnett, Manuel Fahndrich, K. Rustan M. Leino, Peter Mueller, Wolfram Schulte, and Herman Venter. Specification and verification: The spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[10] Rolando Blanco and Paulo Alencar. Categorization of implicit invocation systems. Technical Report CS-2007-31, University of Waterloo, Cheriton School of Computer Science, 200 University Avenue West Waterloo, ON, Canada N2L 3G1, 2007.

[11] Eric Bodden, Éric Tanter, and Milton Inostroza. Joint point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 23, Issue 1*, February 2014.

[12] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[13] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, Berlin, 2006. Springer-Verlag.

[14] Yonghao Chen and Betty H. C. Cheng. A semantic foundation for specification matching. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

[15] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT 2003: Software engineering Properties of Languages for Aspect Technologies at AOSD 2003*, March 2003. Available as Computer Science Technical Report TR03-01a from ftp//:ftp.cs.iastate.edu/pub/techreports/TR03-01/TR.pdf.

[16] David Cok. OpenJML: JML for Java 7 by extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer-Verlag, Berlin, 2011.

[17] Patrick Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proc. SSGRR 2001 – Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, 6 – 10*, 2001.

[18] Frank S. de Boer. A WP-calculus for OO. In Wolfgang Thomas, editor, *Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer-Verlag, 1999.

[19] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag.

[20] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. In *FM 2008: Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 52–67, Berlin, 2008. Springer-Verlag.

[21] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[22] ECMA. *C# language specification*. ECMA Standard 334, February 2006.

[23] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, October 2001.

[24] Robert E. Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns (ECOOP 2001)*, Budapest, Hungary, June 2001. Available from `http://trese.cs.utwente.nl/Workshops/ecoop01asoc/papers/Filman.pdf`.

[25] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, MN, October 2000.

[26] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. Publishing Company, 1993.

[27] David Gorlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. *Lecture Notes in Computer Science*, 551:31–44, 1991.

[28] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012.

[29] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD 04*, pages 26–35, 2004.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[31] Kevin Hoffman and Patrick Eugster. Bridging java and aspectj through explicit join points. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 63–72, New York, NY, USA, 2007. ACM.

[32] Kevin Hoffman and Patrick Eugster. Cooperative aspect-oriented programming. *Sci. Comput. Program.*, 74(5-6):333–354, March 2009.

[33] Kevin Hoffman and Patrick Eugster. Trading obliviousness for modularity with cooperative aspect-oriented programming. *ACM Trans. Softw. Eng. Methodol.*, 22(3):22:1–22:46, July 2013.

[34] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In E. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer-Verlag, 2000.

[35] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[36] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of the 27th International Conference on Software Engineering*, pages 49–58. ACM, 2005.

[37] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.

[38] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005.

[39] Gary T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 2006.

[40] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report 06-20b, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, September 2006.

[41] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[42] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[43] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[44] Carroll Morgan and Ken Robinson. Specification statements and refinement. In Carroll Morgan and Trevor Vickers, editors, *On the Refinement Calculus*, Formal Approaches to Computing and Information Technology (FACIT), pages 23–46. Springer London, 1992.

[45] Joseph M. Morris. Laws of data refinement. *Acta Informatica*, 26(4):287–308, February 1989.

[46] Peter Müller and Joseph N. Ruskiewicz. Rigorous methods for software construction and analysis. chapter A Modular Verification Methodology for C# Delegates, pages 187–203. Springer-Verlag, Berlin, Heidelberg, 2009.

[47] David Notkin, David Garlan, William G. Griswold, and Kevin J. Sullivan. Adding implicit invocation to languages: Three approaches. In *Proceedings of the First JSSST International*

Symposium on Object Technologies for Advanced Software, pages 489–510, London, UK, 1993. Springer-Verlag.

[48] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[49] Thomas Pawlitzki and Friedrich Steimann. Implicit invocation of traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2085–2089, New York, NY, USA, 2010. ACM.

[50] C. Pierik and F.S. de Boer. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs*, 2005.

[51] Cees Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Universiteit Utrecht, 2006.

[52] G. D. Plotkin. Dijkstra's predicate transformers and smyth's powerdomains. In D. Bjorner, editor, *Abstract Software Specifications: 1979 Copenhagen Winter School Proceedings*, volume 86 of *Lecture Notes in Computer Science*, pages 527–553. Springer-Verlag, New York, NY, 1980.

[53] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language of quantified, typed events. Technical Report 07-13a, Iowa State University, Department of Computer Science, October 2007.

[54] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference, Paphos, Cyprus*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179, Berlin, July 2008. Springer-Verlag.

[55] José Sánchez and Gary T. Leavens. Separating obligations of subjects and handlers for more flexible event type verification. In Walter Binder, Eric Bodden, and Welf Löwe, editors,

Software Composition, volume 8088 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag, Berlin, 2013.

[56] José Sánchez and Gary T. Leavens. Static verification of ptolemyrely programs using openjml. In *Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages*, FOAL '14, pages 13–18, New York, NY, USA, 2014. ACM.

[57] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367, New York, NY, October 2007. ACM.

[58] Friedrich Steimann and Thomas Pawlitzki. Types and modularity for implicit invocation with implicit announcement. Obtained from the first author., August 2007.

[59] Kevin Sullivan, William Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *Proc. of the 13th ACM SIGSOFT symposium on the Foundations of software engineering (FSE-13)*, pages 166–175, Lisbon, Portugal, May 2005. ACM Press.

[60] Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with xpis. *ACM Transactions on Software Engineering and Methodology*, 20(2):5:1–5:42, September 2010.

[61] Sun. *JavaBeans*. Sun, August 1997.

[62] Jia Xu, Hridesh Rajan, and Kevin Sullivan. Aspect reasoning by reduction to implicit invocation. In Curtis Clifton, Ralf Lämmel, , and Gary T. Leavens, editors, *FOAL 2004 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004, Lancaster, UK*, num-

ber 04-04 in TR, pages 31–36, Ames, IA, 50011, March 2005. Dept. of Computer Science, Iowa State University.