Reasoning Tradeoffs in Languages with Enhanced Modularity Features

José Sánchez
Universidad Nacional, Costa Rica
jose.sanchez.salazar@una.cr

Gary T. Leavens
University of Central Florida
leavens@cs.ucf.edu

Abstract

The continuous need for more ambitious, more complex, and more dependable software systems demands mechanisms to modularize such systems and reason about their correctness. The reasoning process is affected by the programming language's features, like dynamic dispatching, implicit invocation and oblivious aspect weaving, and by how the programmer uses them. In this paper, by devising a unifying formal setting, we show how reasoning varies with the different language mechanisms, and provide sound rules for reasoning about programs that use these features. While analyzing these mechanisms we explore the main compromises or *tradeoffs* that led to them and explain the disciplines they impose and the strength of the reasoning conclusions one can derive in each case. Our contributions will benefit both language designers and programmers. Language designers will benefit from learning the effects of different modularity features on reasoning. Programmers will learn how to reason about programs that use such features.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules, packages

General Terms Languages, Verification

Keywords Reasoning Tradeoffs, Module Verification

1. Introduction

In large software projects one needs to reason about the behavior of each component in isolation and to reason about the behavior of the entire system. The way in which one can carry out such reasoning tasks is affected by language design decisions regarding composition mechanisms, and by how the programmer uses a language's feature. In this paper we show how reasoning varies with the different mechanisms, and provide sound reasoning rules for them.

In the body of a *client* component there could be various points where *provider* components are invoked. The client component, in combination with the expected behavior of the provider components, must be verified to check that it satisfies its own expected behavior. Different composition mechanisms like dynamic dispatching, implicit invocation and oblivious aspect weaving are variants of this general situation. By devising a unifying formal setting, we analyze and contrast these different mechanisms and show how reasoning varies with them.

Decisions about language features are made by considering compromises or tradeoffs, such as modular vs. nonmodular reasoning, explicit vs. implicit invocation, etc. By deciding on these tradeoffs one can configure different reasoning *scenarios*. Table 1 illustrates some configurations of scenarios and tradeoffs. Each scenario imposes different disciplines, like the well-known *supertype abstraction* discipline [21, 22] for OO scenario, and the authors' coined disciplines of *handler abstraction* and *trigger abstraction* for the Ptolemy scenario [6, 31, 32]. Correspondingly, each scenario allows one to derive different reasoning conclusions.

Table 1: Tradeoffs and Scenarios

Scenarios	Modularity		Announcement		Invocation		Delivery	
(Languages)	mod.	non-mod.	exp.	imp.	exp.	imp.	single	full
OO	×		×		×			
Ptolemy	×		×			×	×	
C# Delegates	×		×			×		×
AO (around)		×		×		×	×	

1.1 Program Reasoning

To *reason about* a program means to conclude, by logical means, some properties of a program's behavior; like its correctness according to certain specification. We write spec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MODULARITY'16, March 14–17, 2016, Málaga, Spain
© 2016 ACM. 978-1-4503-3995-7/16/03...$15.00
http://dx.doi.org/10.1145/2889443.2889447

ifications as triples (P, Q, ε), where P is the precondition, Q the postcondition and ε the frame expression. For asserting that a program S satisfies (or refines) a specification we write $S \sqsupseteq (P, Q, \varepsilon)$. That corresponds to a partial correctness Hoare [16] formula $\{P\}S\{Q\}[\varepsilon]$, stating that whenever S starts execution in a state satisfying P and terminates normally, then the final state satisfies Q and S only modifies state (variables or locations) in ε. The JML [20] program S_1 shown in Figure 1 satisfies its specification as the formula $S_1 \sqsupseteq (x \neq y \wedge x - y \geq -\text{MAX_VALUE}, a > 0, [a, d])$ is valid, where x and y are formal parameters and a and d are global variables. The proof outline of the correctness of S_1 is presented as comments on the source code: starting from the precondition and using the Hoare [16] proof rules of the different language constructs, the postcondition is established.

```
1  //@ requires x!=y && x-y >= -MAX_VALUE;
2  //@ ensures a>0;
3  //@ modifies a,d;
4  S1(int x, int y){
5            // (x≠y ∧ x-y≥-MAX_VALUE)⇒x-y≠0
6   d=x-y; // d≠0 ∧ d≥-MAX_VALUE
7   if(d>0)// d>0
8     a=d; // a>0
9   else //(d≤0 ∧ d≠0 ∧ d≥-MAX_VALUE)⇒-d>0
10    a=-d;// a>0
11 }         // a>0
```

Figure 1: Reasoning about a JML program

Every command in a program is reasoned about using the axiom or proof rule corresponding to that kind of command. In the example in Figure 1 the assignment command (line 6) is reasoned about using the *assignment axiom*, the **`if`** command (lines 7-10) is reasoned about using the *conditional* proof rule, and in this case using the assignment axiom to reason about each branch (lines 8,10). The sequential composition (lines 6-10) of the assignment and the **`if`** commands is reasoned about using the *sequence* rule.

As mentioned before, a key task in developing a complex software system is to decompose it into a set of components and integrate them properly to get the desired functionality. The challenge is not only to reason about isolated components but to reason about their integration.

1.2 Component-Integration Reasoning

Systems are usually assembled out of multiple components. In order to provide its functionality, a main component (*client*) uses or invokes other components (*providers*). In turn, these components use another components and so on. The integration mechanism between components is a key language design decision, where important tradeoffs regarding expressiveness and reasoning must be taken into consideration. For example, in Object-Oriented (*OO*) programming, integration among components is performed by making the client component explicitly call methods from the provider component. Following a modular approach, reasoning about the client code can be performed using the specification, not the implementation, of the called methods.

Figure 2 illustrates the integration of a client component, M, and a provider component, S. The method invocation in the client (line 6) is reasoned about using the specification of the invoked method. The precondition, with actuals substituting formals, must be guaranteed before the invocation and the postcondition can be assumed after.

```
1 //@requires w!=0 && b>0 && -w>=-MAX_VALUE;
2 //@ ensures c>1;
3 //@ modifies a,c,d;
4 M(int w){
5            // w≠0 ∧ b>0 ∧ -w≥-MAX_VALUE
6  S(b,b+w);// b≠b+w ∧ b-(b+w)≥-MAX_VALUE
7            // a>0
8  c=a+1;    // c>1
9 }
```

```
1 //@ requires x!=y && x-y >= -MAX_VALUE;
2 //@ ensures a>0;
3 //@ modifies a,d;
4 S(int x, int y){ ... }
```

Figure 2: OO Component Integration Reasoning

More sophisticated and flexible integration approaches are provided by Implicit Invocation (*II*) [14, 27, 29, 37], in Event Based programming, and Implicit Announcement (*IA*) and Implicit Invocation, in Aspect Oriented (*AO*) [12, 18] programming.

In this paper we identify various important tradeoffs faced when reasoning about OO, II and AO programs, characterize scenarios derived from making choices regarding these tradeoffs, and provide sound proof rules for verification of programs covered by all these scenarios. That not only accounts for a formal treatment of possible reasoning results but also provides guidance for program developers and language designers.

1.3 Benefits for Designers and Programmers

The results of this paper will be useful to both programming language designers and programmers.

Programming language designers need to know how modularity features affect reasoning about programs. Our results will help them make decisions about what features to include in a language, by explaining the effect of these features on reasoning about programs, as well as the discipline that programmers must use to achieve various conclusions.

As presented in Section 4, this work includes a catalog of *reasoning scenarios* that correspond to different implementations of some language features, like *full-delivery* and *single-delivery* implementations of an implicit invocation mechanism. Each scenario is formally solved from a program-reasoning perspective, devising the discipline and reasoning rules that apply.

For example, in an implicit invocation language design, a designer may want to consider whether to use single delivery (where an implicit invocation triggers a single handler that is responsible for calling further handlers), or full delivery (where invocation triggers all handlers in some unspecified order, see Figure 3).

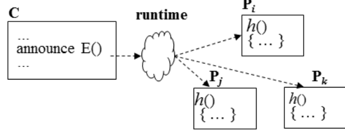


Figure 3: Full Delivery

Our results tell the language designer that with full delivery, each handler must leave the system in a state in which the next handler could be executed. The required discipline is that all handlers satisfy an invariant and the rule is to use that invariant for reasoning about the event announcement in the client. All this is detailed in Section 4.3.1.

In addition to taking advantage of the catalog of reasoning scenarios for the mechanisms we analyze, language designers can also use our unified formal setting to analyze new mechanisms that they create.

Programmers will also benefit from our results. Knowing the required disciplines and reasoning rules for the features, they will be able to write programs to follow the required discipline and thus know how to soundly reason about their programs. For example, when using a full-delivery mechanism, they can select the invariant, check the discipline by verifying each handler against the invariant, and use that invariant to reason about the client.

The paper is organized as follows. Sec. 2 introduces some required definitions and results related to specifications and refinement. Sec. 3 characterizes the identified tradeoffs. Sec. 4 describes the scenarios derived from the tradeoffs and formalizes their proof rules. We review related work in Sec. 5 and offer conclusions in Sec. 6.

2. Specifications and Refinement Preliminaries

For background and to fix notation, we introduce some required definitions and results related to specifications and refinement. In a specification (P, Q, ε), the postcondition Q is a two-state predicate. In Q, left-primed variables $('x)$ represent values in the pre-state, similar to `\old`(x) in JML [20], and normal variables (x) represent values in the post-state.

The *refinement* relation, $\sqsupseteq$, is used to describe both satisfaction between a program and a specification and the strengthening of a specification by a *more-defined* specification, as detailed in Definition 2.1.

DEFINITION 2.1. *(Refinement)*

i. specification refinement (satisfaction) by a program:

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

ii. specification refinement by another specification:

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

This refinement relation between specifications can be characterized as in Theorem 2.2.

THEOREM 2.2 (Refinement Characterization). *The refinement relation ($\sqsupseteq$) between two specifications can be characterized as follows:*

$$(P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon) \Leftrightarrow$$
$$(P \Rightarrow P') \wedge ((old(P) \wedge Q') \Rightarrow Q) \wedge (\varepsilon' \subseteq \varepsilon),$$

where $old(P) = P[\overrightarrow{'x}/\overrightarrow{x}]$ *for* $\overrightarrow{x} = FV(P)$.

The proof can be found in Corollary 5.20 from [21].

While reasoning about a program there could be places in its code where various alternative behaviors may apply. For example, when a program calls a wrapper method, that randomly executes one of two other methods, then either of the two behaviors could apply. The precondition of the two behaviors must be guaranteed and any of the two postconditions can be expected. This is represented by the non-deterministic choice.

DEFINITION 2.3. *(Non-Deterministic Choice)*
The non-deterministic choice ($\Box$) of two specifications is defined as:

$$(P, Q, \varepsilon) \Box (P', Q', \varepsilon') \stackrel{\text{def}}{=} (P \wedge P',\ Q \vee Q',\ \varepsilon \cup \varepsilon')$$

From the characterization of refinement in Theorem 2.2, it is easy to see that the *meet* of two specifications corresponds to their non-deterministic choice [34].

LEMMA 2.4. *(Meet or greatest lower bound of specifications w.r.t. refinement)*

$$(P_j, Q_j, \varepsilon_j) \sqcap (P_k, Q_k, \varepsilon_k) = (P_j \wedge P_k,\ Q_j \vee Q_k,\ \varepsilon_j \cup \varepsilon_k),$$
or in general, $\sqcap_{i=1}^{n} (P_i, Q_i, \varepsilon_i) = (\wedge_{i=1}^{n} P_i, \vee_{i=1}^{n} Q_i, \cup_{i=1}^{n} \varepsilon_i)$

The *meet* of specifications corresponds to a more abstract (*less-defined*) specification, as each one refines their *meet*: $(P_j, Q_j, \varepsilon_j) \sqsupseteq [(P_j, Q_j, \varepsilon_j) \sqcap (P_k, Q_k, \varepsilon_k)]$

It is also easy to compute the *join* of two specifications.

LEMMA 2.5. *(Join or least upper bound of specifications w.r.t. refinement)*

$$(P_j, Q_j, \varepsilon_j) \sqcup (P_k, Q_k, \varepsilon_k) =$$
$$(P_j \vee P_k,\ (old(P_j) \Rightarrow Q_j) \wedge (old(P_k) \Rightarrow Q_k),\ \varepsilon_j \cap \varepsilon_k),$$
or in general,
$$\sqcup_{i=1}^{n} (P_i, Q_i, \varepsilon_i) = (\vee_{i=1}^{n} P_i, \wedge_{i=1}^{n} (old(P_i) \Rightarrow Q_i), \cap_{i=1}^{n} \varepsilon_i)$$

The *join* of specifications corresponds to a more concrete (*more-defined*) specification, as it refines each one of them: $[(P_j, Q_j, \varepsilon_j) \sqcup (P_k, Q_k, \varepsilon_k)] \sqsupseteq (P_j, Q_j, \varepsilon_j)$.

The result in Lemma 2.6 [34] relates refinement and non-deterministic choice.

LEMMA 2.6. *(choice absorption w.r.t. refinement)*

$$(P', Q', \varepsilon') \sqsupseteq (P, Q, \varepsilon) \Leftrightarrow$$
$$((P', Q', \varepsilon') \Box (P, Q, \varepsilon)) \sqsupseteq (P, Q, \varepsilon)$$

Along the lines of stepwise refinement and refinement calculus [3, 4, 24, 25], we consider specifications as a kind of program statement: the *specification statement*. It has the form "**requires** P **ensures** Q **modifies** ε", corresponding to a specification (P, Q, ε). The Hoare formula for the specification statement is

$$\{P\} \ \mathbf{requires}\ P\ \mathbf{ensures}\ Q\ \mathbf{modifies}\ \varepsilon\ \{Q\}[\varepsilon]$$

Its weakest precondition semantics is [28]:

$$wlp(\mathbf{requires}\ P\ \mathbf{ensures}\ Q\ \mathbf{modifies}\ \varepsilon)(R) \stackrel{\text{def}}{=}$$
$$\forall \vec{y} (\forall \vec{u} (P \Rightarrow Q[\vec{y}/\vec{x}]) \Rightarrow R[\vec{y}/\vec{x}])$$

where $\{\vec{x}\}$ represents the program variables, $\{\vec{y}\}$ are fresh auxiliary variables representing the post-state values of $\{\vec{x}\}$, with $\{\vec{y}\} \cap free(P, Q, R) = \emptyset$, and $\{\vec{u}\}$ are the auxiliary variables linking P to Q, with $\{\vec{u}\} = free(P, Q) \setminus \{\vec{x}\}$.

In case the left-primed variables $'\varepsilon$ (representing the values in the pre-state) are the only auxiliary variables linking P and Q, that general formula can be cast as:

LEMMA 2.7. *(wlp of specification statement)*

$$wlp(\mathbf{requires}\ P\ \mathbf{ensures}\ Q\ \mathbf{modifies}\ \varepsilon)(R) =$$
$$\forall \vec{y} ((P \Rightarrow Q[\vec{y}/\vec{\varepsilon},\ \vec{\varepsilon}/\overrightarrow{'\varepsilon}]) \Rightarrow R[\vec{y}/\vec{\varepsilon}])$$

For example, with $R \equiv x \geq 0$ and
$S \equiv$ **requires** $x \neq 5$ **ensures** $x \geq {}'x$ **modifies** $\{x\}$,

$$wlp(S)(R) = \forall y ((x \neq 5 \Rightarrow y \geq x) \Rightarrow y \geq 0) = x \neq 5 \wedge x \geq 0$$

A program can be composed with other programs or specifications by substituting defined fragments or blocks of code in the program by other blocks of code or specification statements. We denote the composition notion by the $\circledcirc$ symbol (for specifications and blocks of code) and by the $\circledast$ symbol (for **proceed** statements).

DEFINITION 2.8. *(Program Composition) If the body b_s of a program S contains certain blocks B_i (for i=1 to n), then the composition of the program with different elements is defined as follows:*

i. (specification composition) Let $(P_i, Q_i, \varepsilon_i)$ be specifications, for i=1 to n, then

$$b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n] \stackrel{\text{def}}{=}$$
$$b_s[\mathbf{requires}\ P_i\ \mathbf{ensures}\ Q_i\ \mathbf{modifies}\ \varepsilon_i / B_i]_{i=1}^n$$

ii. (program composition) Let b_i be blocks of code, for i=1 to n, then

$$b_s \circledcirc [(b_i)_{i=1}^n] \stackrel{\text{def}}{=} b_s[b_i/B_i]_{i=1}^n$$

iii. (proceed composition) Let (P, Q, ε) be a specification, then

$$b_s \circledast [(P, Q, \varepsilon)] \stackrel{\text{def}}{=}$$
$$b_s[\mathbf{requires}\ P\ \mathbf{ensures}\ Q\ \mathbf{modifies}\ \varepsilon / B_i]_{i=1}^n$$

*where the blocks B_i correspond to **proceed** statements in an aspect-oriented or implicit-invocation language.*

It is well known from monotonicity results in the refinement calculus [3, 4, 24, 25] that if $\forall_{i=1\ldots n} (P_i, Q_i, \varepsilon_i) \sqsupseteq B_i$ then $b_s \sqsupseteq (b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n])$ and so if, for example, $(b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n]) \sqsupseteq (P_s, Q_s, \varepsilon_s)$ then $b_s \sqsupseteq (P_s, Q_s, \varepsilon_s)$.

3. Reasoning Tradeoffs

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

3.1 Modular vs. Non-Modular Reasoning

Modular reasoning means being able to establish properties of a module (like a class or aspect) just by considering its interface, specification and implementation, and the interfaces and specifications, but not the implementations, of modules referenced by it. The specification information for each method in a module includes the assumptions it makes (preconditions) and the guarantee (postcondition) it promises [9]. For doing modular reasoning it is required that each method has a specification and also that the references in the invoking module allow one to identify the specifications to use for reasoning about the invocations.

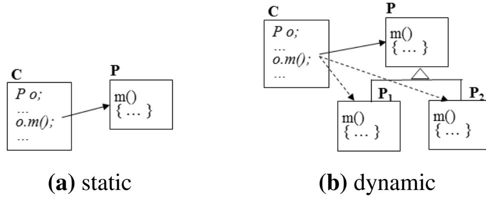
Modular reasoning can be formalized as follows. A *client* method s has a specification $(P_s, Q_s, \varepsilon_s)$ and a body b_s. In the body b_s there are references to *provider* methods $m_1, \ldots, m_n$, each one having its corresponding specification $(P_i, Q_i, \varepsilon_i)$ and body b_i. Modular reasoning about s demands that its body b_s, when composed with the specifications $(P_i, Q_i, \varepsilon_i)$ satisfies s's own specification: $b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n] \sqsupseteq (P_s, Q_s, \varepsilon_s)$. For modular reasoning to be sound, every method 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 methods, then their bodies must 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 methods, 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, since $(b_s \circledcirc [(b_i)_{i=1}^n]) \sqsupseteq (b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n])$, if $b_i \sqsupseteq (P_i, Q_i, \varepsilon_i)$

for all i. Every specification satisfied by the later program is also satisfied by the former: $[(b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i=1}^n]) \sqsupseteq (P_s, Q_s, \varepsilon_s)] \Rightarrow [(b_s \circledcirc [(b_i)_{i=1}^n]) \sqsupseteq (P_s, Q_s, \varepsilon_s)]$, but there could be specifications satisfied by the former and not by the later.

If the invoked method 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 pointcut designators in the aspects select the join points where to apply the advice. Here modular reasoning is difficult, unless obliviousness is relaxed in some way.

In *static dispatching*, as in Figure 4a, each invocation is dispatched to the only one corresponding method implementation in the static type of the target object. In this case modular reasoning can be done exactly as described above.



(a) static **(b)** dynamic

Figure 4: Method Dispatching

In the case of *dynamic dispatching*, as in Figure 4b, 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* [21, 22], 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 about each method invocation using the specification given to that method in the static type of the target object. 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 [21].

3.2 Abstraction vs. Case Analysis

There are situations, like dynamic dispatching and implicit invocation, where a message (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 situation it is not clear what specification to use for reasoning about the message. 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, $(P_i, Q_i, \varepsilon_i) \sqsupseteq (P, Q, \varepsilon)$, and using it to reason about the message, $(call/announce) \sqsupseteq (P, Q, \varepsilon)$. This is sound as the message can be reasoned about as the non-deterministic choice of the individual specifications and this, in turn, refines the abstract specification (as each individual specification refines the abstract one): $(call/announce) \sqsupseteq \square_{i=1}^n (P_i, Q_i, \varepsilon_i) \sqsupseteq (P, Q, \varepsilon)$. One example of this approach is supertype abstractions supported by behavioral subtyping. The specification in the supertype corresponds to the abstract specification that is refined by the specifications in the subtypes.

On the contrary, *case analysis* considers the reasoning requirements for each case separately, without any form of specification inheritance for overwritten methods. The example in Figure 5, adapted from [10], requires a case by case analysis.

```
1  class A{
2   int x; int y;
3   /*@ ensures x==\old(x)+1
4             && y==\old(y)+1; @*/
5   protected 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  protected void inc() {x=x+1;}
13 }
```

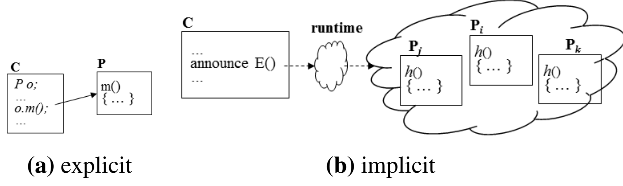
Figure 5: Case Analysis

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`. Nevertheless, both classes are valid as every method in each class satisfies its specification. Method `inc()` satisfies its corresponding specifications in both classes. Method `incX2()` is also valid in both classes since the *requirements* it imposes on method `inc()` (that it at least increments `x`) are part of the *commitments* of this method in both `A` and `Ax`. This illustrates the *abstraction* vs *case analysis* tradeoff.

3.3 Explicit Invocation vs Implicit Invocation

In *explicit invocation* (Figure 6a) 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, behavioral subtyping [2, 21, 23] can be adopted, enabling the use of supertype abstraction for reasoning.

With *implicit invocation* (Figure 6b), the client component announces *events* and the system implicitly invokes the target components, so-called *handlers*, that have been registered for these events.



(a) explicit **(b)** implicit

Figure 6: Invocation

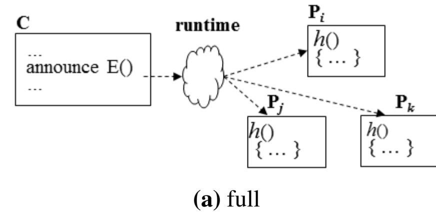
Implicit invocation has many advantages [14]. The loose coupling between components eases system evolution and reuse. Handlers can be added, changed or removed without modifying the client. Independent development is also facilitated. There are also disadvantages. Reasoning about the client is more involved. There are no references to the target components that will be invoked, neither the number of them or their order of execution is known. Some approaches, like the one in Ptolemy [6, 31] and PtolemyRely [32], recover modular reasoning by using event types, that establish the specification that all handlers must satisfy, and using this specification to reason about the event announcements. This *handler abstraction* [34] resembles the supertype abstraction mechanism.

3.4 Full Delivery vs Single Delivery

In event based implicit invocation systems many handlers could be registered for an event. Whether the system directly invokes them all or just one or some is an important tradeoff that must be considered in configuring reasoning scenarios.

In *Full-Delivery* (Figure 7a), upon an event announcement the runtime system invokes all the registered handlers for that event. This strategy is used in some languages, like in C# *delegates* [11, 26] and Java *JavaBeans* [36]. 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. The cases where there are no registered handlers are treated as **`skip`**.

In *single delivery* (Figure 7b) 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 reasoning about each handler, **`proceed`** or **`invoke`** statements need to be considered. This can be done by using the handler's specification to substitute for them: $h_{e_i} \circledast [(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. Again special considerations must be taken when there are no remaining handlers.



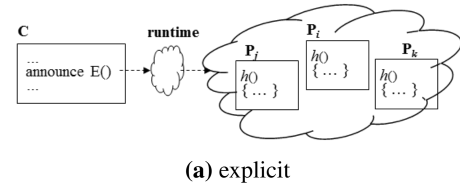
(a) full

(b) single

Figure 7: Delivery Method

3.5 Explicit Announcement vs Implicit Announcement

In general terms, *explicit announcement* (Figure 8a) 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 calls, 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 event announcements are made, one can reason about them as was described in Section 3.3.



(a) explicit

(b) implicit

Figure 8: Announcement Method

With *implicit announcement* (Figure 8b), as in aspect oriented programming, the client code remains oblivious [13] or ignorant of where and what functionality (advice) may be invoked. Different events occurring at certain join points (like jp_a, jp_b and jp_c) 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 (like pc_i, and pc_k), inside aspects, are predicates that select or quantify [13] the join points in the client code, where advice from the aspect is added. Reasoning about the client code requires one to identify the announcing points, selected by the pointcuts, and then reason about these implicit 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 the client. This is particularly useful for implementing cross-cutting concerns. A cross-cutting functionality can be encapsulated in one place, as a piece of advice in an aspect, and applied in many places of a system without changing its source code. However this obliviousness is also a severe disadvantage for reasoning about the client code. There is no clue in it about where and what functionality may be added.

One simple solution proposed [19] 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 determines the advised join points and then a modular reasoning is applied to each one of them. This second phase is the same as reasoning about an implicit invocation system.

4. Reasoning Scenarios and Proof Rules

The tradeoffs presented in the previous section correspond to design decisions faced while developing languages and systems that support OO, II or AO features. By deciding a series of such 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. Once a scenario is configured, appropriate proof rules are provided to reason about systems built using this scenario.

4.1 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 $(P_s, Q_s, \varepsilon_s)$ and a body b_s. In the body b_s there are blocks of code $\{B_i\}$ at which provider modules could be invoked (Fig. 9).

Each block should satisfy its corresponding specification:

$$\{B_i\} \sqsupseteq (P_i, Q_i, \varepsilon_i) \tag{1}$$

The body b_s, using the specifications of the blocks, should satisfy its specification.

$$b_s \circledcirc [(P_i, Q_i, \varepsilon_i)_{i:1..n}] \sqsupseteq (P_s, Q_s, \varepsilon_s) \tag{2}$$

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

Figure 9: Reference Scenario

4.2 The Object Oriented Scenario

As illustrated in Figure 10, in the *Object Oriented* scenario the blocks of the general scenario (Fig. 9) correspond to method invocations, $\{B_i\} \equiv e_i.m_i()$. 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.

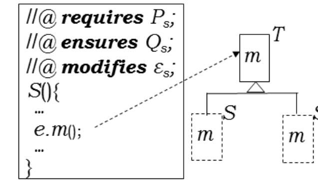


Figure 10: OO Scenario

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 [17, 21, 30], for reasoning about a method invocation, $e.m()$, the proof rule (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 $declOf()$ 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.

$$\text{(OO-INVOKE)} \quad \frac{\Gamma \vdash \boldsymbol{e}{:}T, \quad specOf(T,m) = (P^T_m, Q^T_m, \varepsilon^T_m), \quad declOf(T,m) = T_m\ m(\overrightarrow{T var})}{CT, \Gamma \vdash \begin{array}{l} \{P^T_m[e/this, \vec{e}/\overrightarrow{var}]\} \\ \boldsymbol{e.m(\vec{e})} \\ \{Q^T_m[e/this, \vec{e}/\overrightarrow{var}]\} \\ [\varepsilon^T_m[e/this, \vec{e}/\overrightarrow{var}]] \end{array}} \tag{3}$$

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

$$\forall S \cdot S \leq T \Rightarrow (P^S_m, Q^S_m, \varepsilon^S_m) \sqsupseteq (P^T_m, Q^T_m, \varepsilon^T_m) \tag{4}$$

This discipline is sound but strongly constrains method overriding, requiring behavior-preserving redefinition [10, 21].

4.3 II/EA Scenarios

In the II/EA scenarios the blocks $\{B_i\}$ of the general scenario (Fig. 9) 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.

4.3.1 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. As illustrated in Figure 11, the blocks of the general scenario correspond to event announcements, $\{B_i\}\equiv announce\ e(...)$.

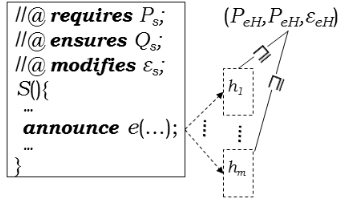


Figure 11: II/EA Full Delivery Mode

For a given announcement of an event, e, the runtime system invokes all the register handlers, h_i, for that event. As pointed out in Section 3.3, for doing modular reasoning in implicit invocation languages *handler abstraction* [34] 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, $h_{i=1...m}$.

$$\forall_{i=1...m} \cdot h_i \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \tag{5}$$

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 (*handler invariance*). That is:

$$Q_{e_H} \Rightarrow P_{e_H} \tag{6}$$

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, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. When there are no registered handlers, the announcement behaves like **skip**, and can be reasoned about by $(P_{e_H}, P_{e_H}, \{\})$. Using (6) and non-deterministic choice definition, it follows that $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_H}, P_{e_H}, \{\}) = (P_{e_H}, P_{e_H}, \varepsilon_{e_H})$, and proof rule (7) puts forth that.

$$\text{(II-FD-ANNOUNCE)} \quad \frac{(\texttt{event e } \{\overrightarrow{Tvar\ sp_H}\}) \in CT, \quad sp_H = \texttt{invariant } P_{e_H} \texttt{ modifies } \varepsilon_{e_H}}{CT, \Gamma \vdash \begin{array}{l} \{P_{e_H}[\vec{e}/\overrightarrow{var}]\} \\ \texttt{announce e(}\vec{e}\texttt{)} \\ \{P_{e_H}[\vec{e}/\overrightarrow{var}]\} \\ [\varepsilon_{e_H}[\vec{e}/\overrightarrow{var}]] \end{array}} \tag{7}$$

The full-delivery mechanism invokes all the registered handlers for an event, ensuring to the programmer that the *added* functionality will be executed. On the other hand, it imposes a general invariant on all handlers, usually allowing only for weak conclusions about the event announcement.

4.3.2 II/EA Single Delivery Scenarios

The idea in single-delivery scenarios is that certain blocks of code in a client program are considered to trigger events, so these blocks are wrapped into **announce** statements. These statements announce the events and pass to the corresponding handlers the triggering block of code, for its eventual execution, as illustrated in Figure 12. The blocks $\{B_i\}$ of the general scenario correspond to event announcements with the announced block of code, $\{B_i\}\equiv announce\ e(...)\{c_i\}$.

The semantics of the **announce** statement is that it executes the first registered handler, if any, or the original block of code, if there are no registered handlers: $announce\ e(\dots)\{C_i\} \equiv h_1 \Box C_i$. Its reasoning, therefore, should consider the non-deterministic choice between a specification satisfied by all handlers, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and the specification, $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})$, of the particular announced block of code, c_i:

$$(\texttt{announce } e(\dots)\{c_i\}) \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{B_i}, Q_{B_i}, \varepsilon_{B_i}) \tag{8}$$

This block, c_i, is known because it is part of the statement being reasoned about.

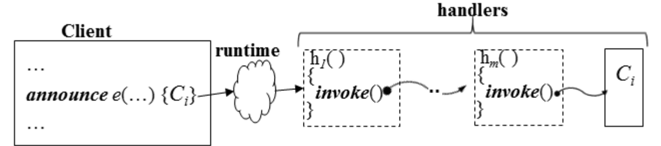


Figure 12: II/EA Single Delivery at Runtime

Invoke statements, in the body of a handler, execute the next handler, if any, or the passed-on block of code, otherwise: $invoke() \equiv h_{j+1} \Box C_i$. Their reasoning should consider the non-deterministic choice between a specification satisfied by all next handlers, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and a specification satisfied by all announced blocks for that event, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$. In this case the block, C_i, is also unknown, as the invocation could be in the execution chain for any announcement for the corresponding event.

$$(\texttt{invoke}()) \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \Box (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}) \tag{9}$$

For doing modular reasoning in single delivery scenarios one requires a specification that abstracts all the handlers for an event (*handler abstraction* [34]) and a specification that abstracts all the triggering blocks of code for that event (*trigger abstraction* [34]). 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 13.

The behavior of each triggering block of code C_i in the original program is characterized by a corresponding specification $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})$.

Trigger abstraction [34] requires one to choose a base-code specification $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$ that generalizes all of them:

$$\forall_{i=1...n} \bullet \Big(C_i \sqsupseteq (P_{B_i}, Q_{B_i}, \varepsilon_{B_i})\Big) \wedge \Big((P_{B_i}, Q_{B_i}, \varepsilon_{B_i}) \sqsupseteq (P_{e_B}, Q_{e_B}, \varepsilon_{e_B})\Big) \tag{10}$$

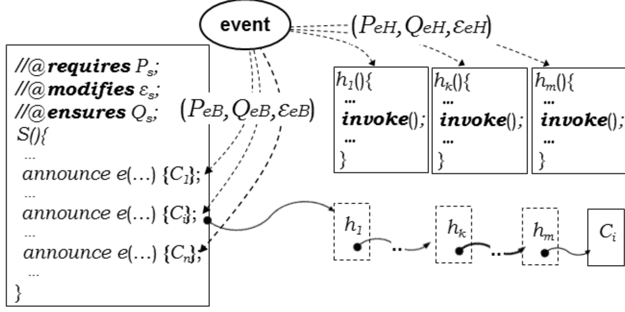


Figure 13: II/EA Single Delivery Abstraction

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, whose value was established by lemma 2.4.

$$(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})= \sqcap_{i=1}^{n}(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})=(\wedge_{i=1}^{n}P_i, \vee_{i=1}^{n}Q_i, \cup_{i=1}^{n}\varepsilon_i) \quad (11)$$

Handler abstraction [34] 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 handler's specification.

$$\forall_{i=1\ldots m} \bullet [h_i \circledast ((P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \,\square (P_{e_B}, Q_{e_B}, \varepsilon_{e_B}))] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H}) \quad (12)$$

Different ways to establish the handlers' specification lead to different scenarios.

4.3.3 Ptolemy Single-Delivery Scenario

The Ptolemy [6, 31] language uses the single-delivery strategy. In Ptolemy event types are declared independently from the client components, which announce them, and from the provider components that handle them. The event declaration includes the specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, that both handlers and announcing blocks of code for the event must satisfy, as shown in Figure 14.

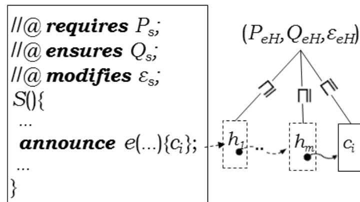


Figure 14: Ptolemy Scenario

Each announced block, C_i, behaves like a handler and is modularly checked to refine the event specification, $C_i \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. Taking $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})=(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$ it follows that $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})\square(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})$ yields $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and so, according to (8), event **announce** statements can be reasoned about using the specification $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$ declared in the event, yielding the proof rule in (13).

(II/EA-PTOLEMY-ANNOUNCE)

$$\frac{(\textbf{event}\ e\ \{\overrightarrow{T\ var\ sp_H}\})\in CT, \quad sp_H = \textbf{requires}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\ \textbf{ensures}\ Q_{e_H}}{CT, \Gamma \vdash \begin{array}{l}\{P_{e_H}[\overrightarrow{e}/\overrightarrow{var}]\} \\ \quad \textbf{announce}\ e(\overrightarrow{e})\{C_j\} \\ \{Q_{e_H}[\overrightarrow{e}/\overrightarrow{var}]\} \\ \left[\varepsilon_{e_H}[\overrightarrow{e}/\overrightarrow{var}]\right]\end{array}} \quad (13)$$

For reasoning about **invoke** statements the following applies. As $(P_{B_i}, Q_{B_i}, \varepsilon_{B_i})=(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$ then, by (11), it follows that $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})=(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$. So $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})\square(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})=(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and thus, according to (9), **invoke** statements can also be reasoned about using the specification $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$ declared in the event, leading to the following proof rule:

(II/EA-PTOLEMY-INVOKE)

$$\frac{(\textbf{thunk}\ e)=\Gamma(n), \quad (\textbf{event}\ e\ \{\overrightarrow{T\ var\ sp_H}\})\in CT, \quad sp_H = \textbf{requires}\ P_{e_H}\ \textbf{modifies}\ \varepsilon_{e_H}\ \textbf{ensures}\ Q_{e_H}}{CT, \Gamma \vdash \begin{array}{l}\{P_{e_H}[\overrightarrow{n.var}/\overrightarrow{var}]\} \\ \quad n.\textbf{invoke}() \\ \{Q_{e_H}[\overrightarrow{n.var}/\overrightarrow{var}]\} \\ \left[\varepsilon_{e_H}[\overrightarrow{n.var}/\overrightarrow{var}]\right]\end{array}} \quad (14)$$

The handlers are reasoned about using handler abstraction (12) which, using the fact that $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})\square(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})=(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, corresponds to $[h_i \circledast (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})] \sqsupseteq (P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, allowing again modular reasoning.

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 approach incomplete, as there exist valid programs that cannot be verified [32].

4.3.4 PtolemyRely Single-Delivery Scenario

By constrast, in PtolemyRely language [32], which is an extension of Ptolemy [6, 31], the handler's specification is independent of the base-code specification (Figure 15). Event declarations includes both: a handler's specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, and an announced-code specification, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$.

Assuming *trigger abstraction* (10) and *handler abstraction* (12), reasoning rules can be formulated. According to (8), **announce** statements are reasoned about as the non-deterministic choice between the handlers and the current

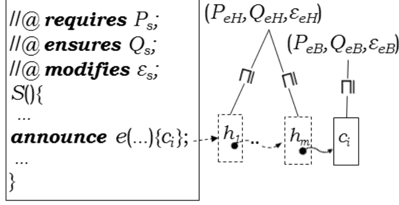


Figure 15: PtolemyRely Scenario

announced-code.

(II/EA-PtolemyRely-Announce)

$$\frac{\begin{array}{c}(\texttt{event}\ e\ \{\overrightarrow{T\ var}\ \texttt{relies}\ sp_B\ sp_H\})\in CT,\\ sp_B{=}\texttt{requires}\ P_{e_B}\ \texttt{modifies}\ \varepsilon_{e_B}\ \texttt{ensures}\ Q_{e_B},\\ sp_H{=}\ \texttt{requires}\ P_{e_H}\ \texttt{modifies}\ \varepsilon_{e_H}\ \texttt{ensures}\ Q_{e_H},\\ CT,\Gamma\vdash\{P_{B_j}\}\ C_j\ \{Q_{B_j}\}[\varepsilon_{B_j}]\end{array}}{CT,\Gamma\vdash\begin{array}{c}\{P_{e_H}[\overrightarrow{e}/\overrightarrow{var}]\wedge P_{B_j}\}\\ \texttt{announce}\ e(\overrightarrow{e})\{C_j\}\\ \{Q_{e_H}[\overrightarrow{e}/\overrightarrow{var}]\vee Q_{B_j}\}\\ \left[\varepsilon_{e_B}[\overrightarrow{e}/\overrightarrow{var}]\cup\varepsilon_{B_j}\right]\end{array}} \quad (15)$$

According to (9), **invoke** statements can be reasoned about as the non-deterministic choice between the handler's specification and the base-code specification.

(II/EA-PtolemyRely-Invoke)

$$\frac{\begin{array}{c}(\texttt{thunk}\ e){=}\Gamma(n),\\ (\texttt{event}\ e\ \{\overrightarrow{T\ var}\ \texttt{relies}\ sp_B\ sp_H\})\in CT,\\ sp_B{=}\texttt{requires}\ P_{e_B}\ \texttt{modifies}\ \varepsilon_{e_B}\ \texttt{ensures}\ Q_{e_B},\\ sp_H{=}\ \texttt{requires}\ P_{e_H}\ \texttt{modifies}\ \varepsilon_{e_H}\ \texttt{ensures}\ Q_{e_H}\end{array}}{CT,\Gamma\vdash\begin{array}{c}\{(P_{e_H}\wedge P_{e_B})[\overrightarrow{n.var}/\overrightarrow{var}]\}\\ n.\texttt{invoke}()\\ \{(Q_{e_H}\vee Q_{e_B})[\overrightarrow{n.var}/\overrightarrow{var}]\}\\ \left[(\varepsilon_{e_H}\cup\varepsilon_{e_B})[\overrightarrow{n.var}/\overrightarrow{var}]\right]\end{array}} \quad (16)$$

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 one to have separate specifications for handlers and subjects (base code) [32]. However, the base-code reasoning is weakened since reasoning about **announce** statements is weaker than reasoning about the corresponding original code, as clearly $(P_{B_j}, Q_{B_j}, \varepsilon_{B_j})\sqsupseteq (P_{e_H}\wedge P_{B_j},\ Q_{e_H}\vee Q_{B_j},\ \varepsilon_{e_H}\cup\varepsilon_{B_j})$.

4.4 AO Scenarios

In AO scenarios the blocks $\{B_i\}$ in the reference scenario (Figure 9) correspond to shadows of join points, that is, pieces of code considered to trigger the join-point events [15]. AO scenarios are 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 [37]. The pointcut declaration in AO corresponds to the event declaration in II. The shadows in AO correspond to the blocks of code that announce events in II. The pieces of advice correspond to the handlers in II. The *proceed* instruction in AO corresponds to the *invoke* instruction in Ptolemy language.

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, C, in the base code all matching pointcuts, e_i, are identified, and all the pieces of advice for those pointcuts are considered the handlers for the join-point event. Then, reasoning strategies similar to the ones 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, as shown in Figure 16. A pointcut specification, e, declares the specification, $(P_{e_B}, Q_{e_B}, \varepsilon_{e_B})$, that matching join point shadows must satisfy, and the specification, $(P_{e_H}, Q_{e_H}, \varepsilon_{e_H})$, implementing pieces of advice should refine, as in PtolemyRely.

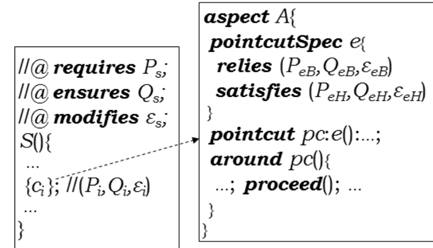


Figure 16: AO Scenario

The specification, $(P_i, Q_i, \varepsilon_i)$, for each shadowed join-point depends on the type of join-point. 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.

The typical scenario in Figure 17 allows one to illustrate the reasoning rules.

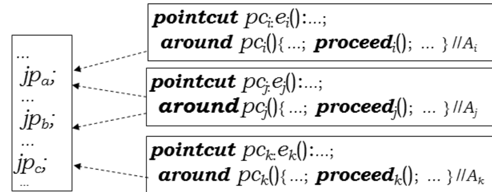


Figure 17: AO Reasoning

Every joint-point satisfies its specification and this must refine the *relies* specification in every matching pointcut. For example, in Figure 17, jp_a refines its specification, $jp_a\sqsupseteq (P_a, Q_a, \varepsilon_a)$, and it refines the specification in the matching pointcuts pc_i and pc_j, $(P_a, Q_a, \varepsilon_a)\sqsupseteq(P_{e_{Bi}}, Q_{e_{Bi}}, \varepsilon_{e_{Bi}})$ and $(P_a, Q_a, \varepsilon_a)\sqsupseteq(P_{e_{Bj}}, Q_{e_{Bj}}, \varepsilon_{e_{Bj}})$. After weaving, the joint-points are reasoned about using their original specification, in case no advice applies, or the non-deterministic choice of the *satisfies* specifications for all the pointcuts that apply. In a piece of advice, **proceed** instructions may execute the body of any join-point matching its pointcut, all abstracted by the *relies* specification, or the body of any other piece of advise advising the same join-point, each one abstracted by its corresponding *satisfies* specification. In the example, $proceed_j\equiv jp_a\Box jp_b\Box A_i\Box A_j$, so $proceed_j\sqsupseteq$

$(P_{e_{Bj}}, Q_{e_{Bj}}, \varepsilon_{e_{Bj}}) \Box (P_{e_{Hi}}, Q_{e_{Hi}}, \varepsilon_{e_{Hi}}) \Box (P_{e_{Hj}}, Q_{e_{Hj}}, \varepsilon_{e_{Hj}})$. The body of each piece of advice must be reasoned about against its corresponding *satisfies* specification proceed-composed with the computed *proceed* specification. This scenario is the most flexible one but requires a non-modular, whole-program, reasoning step.

5. Related Work

In this paper we present part of the work of author J. Sánchez in his dissertation [34]. That work was also inspired by previous work on Ptolemy language [6, 31] and its extensions in PtolemyRely [32, 33]. Ptolemy is an extension of Java with support for the implicit invocation architectural style [14, 27], plus some aspect oriented features [12, 18], noticeably *around*-like advice (handlers) and *proceed*-like invocations (*invoke* statement). It also incorporates translucid contracts [5, 6] that provide functional specification features and control effect reasoning.

Between many others approaches similar to Ptolemy, Joint Point Interfaces (JPI) [8] and Open Modules (OM) [1] have also been proposed as a mechanism for decoupling aspects from base code. JPI extends and refines the notion of join point types (JPT) [35]. Implicit announcement is supported through polymorphic pointcuts, and explicit announcement through closure join points. Both JPI and JPT lack specification and verification features like the ones in Ptolemy, providing that could be an interesting future work. OM is also similar to Ptolemy but the quantification mechanism is different, so a reasoning scenario similar to Ptolemy's should be applicable.

Blanco and Alencar [7] presented a categorization of implicit invocations systems in which they enumerate some "key properties of interest", or tradeoffs, to be considered while designing this type of systems. Their work combines previous work from Notkin et al.[27], on design choices when extending traditional languages with implicit invocation, and from Meier and Cahill [27], on a taxonomy of event-based programming.

The distinctive contribution of our work is our focus on how the different tradeoffs affect the way a system can be reasoned about.

6. Conclusions

This paper shows how reasoning varies with different language mechanisms. Disciplines required by different mechanisms, have been devised and sound reasoning rules have been provided for each mechanism, creating a partial catalog of reasoning scenarios. Important tradeoffs faced while shaping this mechanisms were considered and a general unified model that subsumes all of them was developed. The catalog of reasoning scenarios is valuable for both language designers and programmers and our unified model could be used to analyze scenarios derived from new mechanisms.

Besides providing proof rules for different scenarios and showing their rationale, some general observations can be made. Modular reasoning is preferred to whole program reasoning as it better tackles the challenge of developing big and complex software systems. Full delivery of events, as in C# and JavaBeans, enforces the invocation of all handlers but impose an invariant on all of them. Explicit announcement with implicit invocation, as in Ptolemy and PtolemyRely, is attractive as it allows one to do modular reasoning. Aspect oriented reasoning can be tackled by using implicit-invocation like reasoning, but requiring a global pre-processing step [19].

Acknowledgments

The work of both authors was partially supported by US NSF under grant CCF-1017334. The work of José Sánchez is also supported by Costa Rica's Universidad Nacional (UNA) and Ministerio de Ciencia y Tecnología (MICIT).

References

[1] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. Available from http://www-2.cs.cmu.edu/~aldrich/aosd/, 2003.

[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] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.

[4] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, Berlin, 1998.

[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, 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] ECMA. *C# language specification*. ECMA Standard 334, February 2006.

[12] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, October 2001.

[13] 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.

[14] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. *Lecture Notes in Computer Science*, 551:31–44, 1991.

[15] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD 04*, pages 26–35, 2004.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *TOPLAS*, 37(4):13:1–13:88, August 2015.

[22] 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.

[23] 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.

[24] 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.

[25] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[26] 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.

[27] 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.

[28] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

[29] 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.

[30] Cees Pierik. *Validation Techniques for Object-Oriented Proof Outlines*. PhD thesis, Universiteit Utrecht, 2006.

[31] 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.

[32] 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.

[33] 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.

[34] Jose A. Sanchez. *Reasoning Tradeoffs in Implicit Invocation and Aspect Oriented Languages*. PhD thesis, Univ. of Central Florida, Dept. of Electrical Engineering and Computer Science, 2015. Technical Report CS-TR-15-02.

[35] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, pages 1:1–1:43, 2010.

[36] Sun. *JavaBeans*. Sun, August 1997.

[37] 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*, number 04-04 in TR, pages 31–36, Ames, IA, 50011, March 2005. Dept. of Computer Science, Iowa State University.