Applying Translucid Contracts for Modular Reasoning about Aspect and Object Oriented Events

Mehdi Bagherzadeh[β] Gary T. Leavens[θ] Robert Dyer[β]
[β]Iowa State University [θ]University of Central Florida
{mbagherz, rdyer}@iastate.edu leavens@eecs.ucf.edu

ABSTRACT

The Implicit Invocation (II) architectural style improves modularity and is promoted by aspect-oriented (AO) languages and design patterns like Observer. However, it makes modular reasoning difficult, especially when reasoning about control effects of the advised code (subject). Our language Ptolemy, which was inspired by II languages, uses translucid contracts for modular reasoning about the control effects; however, this reasoning relies on Ptolemy's event model, which has explicit event announcement and declared event types. In this paper we investigate how to apply translucid contracts to reasoning about events in other AO languages and even non-AO languages like C#.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Programming by contract, Assertion checkers; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions, Invariant, Pre- and post-conditions, Specification techniques

General Terms

Design, Languages, Verification

Keywords

Translucid contracts, modular reasoning, implicit invocation, aspect-oriented interfaces, grey-box specification, Ptolemy, quantified typed events, aspect-oriented events, object-oriented events

1. INTRODUCTION

Reasoning about the control effects of aspect-oriented (AO) programs seems difficult because: (1) join point shadows are pervasive, and (2) advice can have interesting control effects (e.g., throwing an exception or not proceeding) which are difficult to specify using black-box behavioral contracts. One way to avoid the first problem is to limit the application of advice to the base code. In our previous work on Ptolemy, join point shadows are limited to the places where events are explicitly announced [13]. To solve the second problem, we proposed translucid contracts [3]; these are grey-box based specifications limiting the behavior of advice. The grey-box nature of translucid contracts makes it possible to reveal some implementation details while hiding others.

In this paper we show the extent to which translucid contracts can be applied to several AO interface proposals as well as a non-AO language (C#). That is, we separate the ideas of translucid contracts from their original context, namely the Ptolemy language. The key features of Ptolemy that are relevant are explicitly declared event types, explicit event announcement and its quantification mechanism. Ptolemy's event announcement makes join point shadows in the base code, explicit. The quantification mechanism allows static computation of the set of advice at a specific place in the code.

Contributions of this work include:

- Application of translucid contracts to other AO interfaces, specifically crosscutting programming interfaces (XPI) [17], aspect-aware interfaces (AAI) [9] and Open Modules [1].
- A programming idiom to apply translucid contracts to a non-AO language with built-in support for events, C#.

In the rest of the paper, Section 2 provides background information about translucid contracts in Ptolemy. Section 3 shows how to apply translucid contracts to other proposals for AO interfaces. Section 4 discusses a proposed programming idiom to apply translucid contracts to C# events. Section 5 discusses related work and finally Section 6 concludes the paper.

2. TRANSLUCID CONTRACTS IN PTOLEMY

The canonical figure editor example in Figure 1, illustrates translucid contracts in the Ptolemy language [13]. A figure element `Point` sets the value of its x-coordinate in method `setX`. The requirement in this example is: skip the modification of the x-coordinate, of the figure element point, if the figure element is *fixed* and not modifiable. This requirement could be implemented using event-driven programming techniques, which announce an event when `setX` is about to modify the `Point` and have an event handler method like `enforce` which enforces the non-modifiability requirement of the fixed figure element.

Our language Ptolemy, used in the implementation of the example in Figure 1, enables event-driven programming by the introduction of *quantified, typed events*. Event type `Changed` (lines 10-20) abstracts concrete events which represent modification to figure elements, such as points. Context variable `fe` (line 11) is a piece of information communicated between `Point` (subject), which announces `Changed`, and its handler `Enforce` (observer). The

FOAL'11, March 21, 2011, Pernambuco, Brazil.

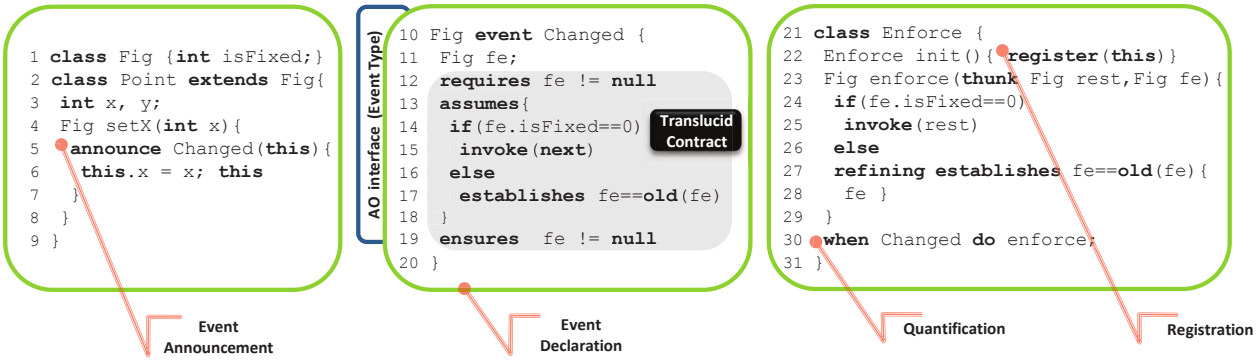

Figure 1: A translucid contract for the event type `Changed`

translucid contract (lines 12–19) limits the behavior of the refining handler methods like `enforce` using pre- and post-condition constraints phrased in **`requires`** and **`ensures`** clauses (lines 12 and 19). It also limits the control effects of the refining handlers by imposing structural constraints on their implementation using **`assumes`** block (lines 13–18). Subject `Point` announces event `Changed` explicitly using an **`announce`** expression (lines 5–7), passing the parameter **`this`** to be mapped to the context variable `fe`. Observer `Enforce` shows its interest in being notified about announcements of event `Changed` using the binding declaration **`when`** – **`do`** (line 30), which says to run method `enforce` whenever an event of type `Changed` is announced. The subject `Enforce` registers itself as an observer for event `Changed` using the **`register`** expression (line 22).

As mentioned earlier, translucid contracts restrict the control effects of the refining handlers by imposing constraints on the structure of the code in their implementation. Handlers of a specific event should refine the translucid contract of the event. The **`assumes`** block (lines 13–18) contains this information. Translucid contracts are more expressive compared to black-box contracts as they can reveal some implementation details about their refining handlers using *program expressions*, while hiding others using *specification expressions*. For example, the program expression (line 14) is conveying the fact that each refining handler must evaluate the **`if`** expression in its implementation as the very first expression followed by an **`invoke`** (line 15). While program expressions reveal implementation details, specification expressions (line 17) hide them, which allows for variability in the refining handlers' implementations. The programmer of the observer module, by just looking at the observer and the translucid contract, can conclude that if the figure element `fe` is not fixed then the handler method is called, allowing the modification of the figure (lines 14–15); otherwise the handler is skipped and the figure is not changed (line 17). **`invoke`** is Ptolemy's equivalent of AspectJ's **`proceed`**.

In terms of variability of the handlers, outside the scope of this example, structural constraints in the assumes block could be as liberal as **`establishes`** $true$ which specifies any handler without an invoke expression in its body or **`establishes`** $true$; **`invoke`**(**`next`**); **`establishes`** $true$ which allows any handler, with the invoke expression somewhere in its implementation.

Verification of the handler method's refinement of the translucid contracts is carried out via a hybrid static and dynamic approach. Static structural refinement checks for the textual matching between program expressions in the translucid contract and the handler implementation at the same structural positions in the code and the contract [3]. For example, lines 14–16 match lines 24–26. Specification expressions in the contract must be refined by **`refining`** expressions carrying the same specification. For example, line 17 is refined by the refining expression on lines 27-28. Runtime assertions assure that refining expressions actually refine the specification they claim to refine. Pre- and post-conditions of the translucid contract are also enforced using runtime probes inserted at the beginning and end of each handler and before and after event announcement.

The key point to notice when applying translucid contracts to the event types in Ptolemy, is that: *In Ptolemy, each handler knows about the type of events it handles, statically at compile time*. Thus, having the handler's implementation and the declaration of the event type it handles, refinement of the contract by the handler could be carried out modularly without any need for whole-program analysis. This is not the case in all languages with built-in event-driven mechanism such as C#. In these languages *handlers do not statically know about the type of events they might handle.* In this work, we propose a very simple programming idiom which allows the handlers to know about the type of events they handle, which in turn enables modular verification of their refinement of the translucid contract of the events they handle.

3. APPLICABILITY TO OTHER AO INTERFACES

As mentioned in Section 1, pervasive join point shadows are one of the obstacles in the modular reasoning about AO programs. AO interfaces tackle this problem by making join points explicit. Ptolemy's event types could be thought of as AO interfaces. We show the applicability of translucid contracts to crosscutting interfaces (XPI) [17], aspect-aware interfaces (AAI) [9], and Open Modules [1] and discuss changes in the refinement rules required to verify such programs. Other AO interfaces such as join point types (JPT) [16] and explicit join points (EJP) are not discussed as they are similar to Ptolemy's event types, discussed in our previous work [3]. For a more detailed discussion on the applicability of translucid contracts to AO interfaces see our previous work [2].

3.1 Translucid Contracts for XPIs

The key idea in crosscut programming interfaces (XPIs) [17] is to establish an interface, based on design rules, to decouple the base and the aspect design. An XPI limits the exposure of join points and also the behavior of advised and advising code using black-box contracts in terms of provides and requires clauses, with no mechanism to check the full compliance to the contract.

Figure 2 illustrates the applicability of translucid contracts to XPI `Changed` on lines 4–11, in an AspectJ implementation of the figure editor example introduced in Section 2. XPI `Changed`

```
1 aspect Changed {
2 pointcut jp(Fig fe):
3  call(Fig Fig+.set*(..))&& target(fe);
4 requires fe != null
5 assumes{
6  if(fe.fixed == 0)
7   proceed(fe);
8  else
9   establishes fe == old(fe);
10  }
11 ensures  fe != null
12 }
```

```
13 aspect Enforce {
14  Fig around(Fig fe): Changed.jp(fe){
15   if(fe.fixed == 0)
16    proceed(fe);
17   else
18    refining establishes fe==old(fe){
19     return fe;
20   }
21  }
22 }
```

Figure 2: Applying translucid contract to XPI

and aspect `Enforce` in Figure 2 are the counterparts of Ptolemy's event type `Changed` and handler `Enforce` in Figure 1. The language for expressing translucid contracts is slightly adapted to use AspectJ's **`proceed`** instead of Ptolemy's **`invoke`**, on lines 7, 16.

Unlike Ptolemy, where the translucid contract is attached to the event type (lines 12–19, Figure 1), in the XPI the contract is attached to the pointcut declaration (lines 4–11, Figure 2). In the Ptolemy example of Figure 1 only the context variable `fe` defined on line 11 could be accessed in the contracts, likewise in the XPI example, only the variable `fe` exposed by the pointcut (lines 2–3, Figure 2) is used in the contract. In Ptolemy the event type of interest is specified by the handler in the binding declaration (line 30, Figure 1) whereas in the XPI example, handler `Enforce` reuses the pointcut declaration in XPI `Changed` (line 14, Figure 2). Our refinement rules could be added here in the AO type system enforcing that the advice body on lines 15–21 must refine the translucid contract of the pointcut declaration on line 14. As it can be seen, the refinement rules are applicable to XPIs with only minor changes.

3.2 Translucid Contracts for AAIs

Some AO interfaces such as XPIs could be specified explicitly, whereas others such as aspect-aware interfaces (AAIs) [9] could be computed from the implementation, given whole-program information. Figure 3 illustrates the AAI for the figure editor example of Section 2. Figure 3 shows the extracted AAI for the method `setX` on lines 3–4 along with a translucid contract on lines 5–12, carried over from the pointcut to the join point shadow. In AAI the advised join point in method `setX` contain the details of the advising advice on lines 3–4. Syntax and refinement rules similar to XPIs are applicable here. Similar ideas can also be applied to aspect-oriented development tools such as AJDT, which provide AAI-like information at each join point shadow in an AspectJ program.

3.3 Translucid Contracts for Open Modules

Open Modules [1] allow explicit exposure of pointcuts for behavioral modifications by aspects, which is similar to signaling events using the announce expression in the Ptolemy. The implementations of these pointcuts remain hidden from the aspects which in turn reduces the impact of the base code changes on the aspect. However, in Open Modules, each explicitly declared pointcut has to be enumerated by the aspect for advising.

```
1 class Point extends Fig {
2  int x, y;
3  Fig setX(int x): Enforce -
4   after returning Changed.jp(Fig fe)
5    requires fe != null
6    assumes{
7     if(fe.fixed == 0)
8      proceed(fe);
9     else
10     establishes fe == old(fe);
11   }
12   ensures  fe != null
13 /* body of setX */
14 }
```

Figure 3: Applying translucid contract to AAI

```
1 module Changed{
2  class Fig;
3  expose to Enforce: call(Fig Fig+.set*(..));
4  requires fe != null
5  assumes{
6   if(fe.fixed == 0)
7    proceed(fe);
8   else
9    establishes fe == old(fe);
10  }
11  ensures  fe != null
12 }
```

```
13 aspect Enforce {
14  Fig around(Fig fe): target(fe) &&
15   call(Fig Fig+.set*(..));
16   if(fe.fixed == 0)
17    proceed(fe);
18   else
19    refining establishes fe==old(fe){
20     return fe;
21   }
22  }
23 }
```

Figure 4: Applying translucid contract to Open Modules

Figure 4 illustrates the applicability of translucid contracts, lines 4–11, to Open Module `Changed` in the figure editor example of Section 2. To retain similarity with other examples in the paper, syntax from Ongkingco *et al.*'s AspectJ implementation [12] is used in the example. Compare Open Module `Changed` and aspect `Enforce` with event type `Changed` and handler `Enforce` in Figure 1. Open Module `Changed` in Figure 4 exposes a pointcut of **`class`** `Fig` on line 2 which is only advisable by the aspect `Enforce` marked by **`expose to`**, line 3. The translucid contract on lines 4–11 limits the the interaction between `Enforce` and the pointcut exposed on line 3.

Like contracts in XPIs, in Open Modules the contract on lines 4–11 is attached to the pointcut declaration on line 3. Variable `fe` named in the contract is the one exposed by the pointcut on line 3, again like XPIs. The proposed rules for verifying refinement need to be modified slightly. In Ptolemy, the event type of interest `Changed` is specified in the binding declaration (line 30, Figure 1), whereas in the AspectJ implementation of Open Modules [12], aspects cannot reuse pointcuts exposed by the Open Module and need to enumerate the pointcut in the advice declaration again, lines 14–15. Refinement rules could be added here in the AO type system. The same adaptations in the syntax and refinement rules as of XPI's are applicable to Open Modules. The challenge is to match aspect `Enforce` pointcut definition on lines 14–15, with the Open Module one on line 3 to pull out its contract for refinement checking.

4. APPLICABILITY TO NON-AO LANGUAGES

Section 3 discussed the application of translucid contracts to AO interfaces rather than Ptolemy's event types. But the applicability of translucid contracts is not limited to just AO languages. In this section we discuss their applicability to a non-AO language, C#, with built-in support for event announcement and handling.

4.1 Problem

As discussed earlier in Section 1, Ptolemy's key feature for applicability of translucid contracts is that for any specific handler the set of potential events it handles is statically known. In other words, for each event type in Ptolemy, it is pretty straightforward to determine the set of its potential handlers using Ptolemy's quantification mechanism. Thus the translucid contract for the handler could be easily pulled out and refinement can be checked in a modular fashion using only the handler implementation and the contract.

In languages with built-in event announcement and handling, such as C#, the set of handlers for an event is not easily known statically. In C# the event model relies on type-safe method pointers (delegates) which could be used to dynamically register a method as a handler for a specific event. The signature of the handler often only includes the context variable and does not indicate the specific type of event being handled, such as:

```
Fig enforce (Fig fe);
```

This handler could handle multiple events, as long as the events pass in the context variable `fe` of type `Fig`. To determine the specific event being handled by each handler, we propose a simple programming idiom which *requires the event type to be passed as an argument to the handler method*. Using this idiom, by only looking at the handler method's signature, the type of event it handles can be easily determined. The idiom resembles the quantification mechanism in Ptolemy, as in line 30 in Figure 1.

4.2 Translucid Contracts for C#

In this section event declaration, announcement and handling in C# is illustrated and compared with Ptolemy using the figure editor example in Figure 1. The C# example is more verbose than needed in order to provide handlers with an **`Invoke`** statement which causes the next applicable handler to run, like its counterpart the invoke expression in Ptolemy. This section also discusses the proposed programming idiom. All our proposal requires is to pass into the handler the event type it handles, as a formal parameter.

```
10 class Changed:EventType <Fig, Changed.Context> {
11  class Context{
12   Fig fe;
13   Context (Fig fe){ this.fe = fe;}
14   Fig contract() {
15    Contract.Requires(fe != null);
16    Contract.Ensures(fe != null);
17    if (fe.isFixed==0)
18     return new Changed().Invoke();
19    else {
20     Contract.Assert(1==1);
21     Contract.Assert(fe==Contract.OldValue(fe));
22    }
23  }}}
```

Translucid Contract

Figure 5: Applying translucid contract to C#

Figure 5 illustrates declaration of event type `Changed`, similar to `Changed` in Figure 1, with return type `Fig`, line 10, and the context variable `fe`, defined on line 12 and set on line 13. Like Ptolemy, in C# the contracts are attached to the event type, lines 15–21. Method `contract` on lines 14–22 is the placeholder for the translucid contract. Lines 15–16 state pre- and post-conditions of the contract using the Embedded Contracts Language [6]. Lines 17–22 illustrate the body of the **`assumes`** block of Figure 1 lines 13–18. Lines 20–21 in Figure 5 are the equivalent of the specification expression of line 17 in Figure 1. Specification **`establishes`** fe == **`old`** (fe) is the sugar for **`requires`** $true$ **`ensures`** fe == **`old`** (fe). The **`Invoke`** method on line 18 causes the next applicable handler to run. It is provided by the class `EventType` in the C# library for Ptolemy, which is not shown here.

```
1 class Fig { int isFixed; }
2 class Point:Fig {
3  int x, y;
4  void setX(int x) {
5   Changed.Announce(new Changed.Context(this),()=>{
6    this.x = x;
7    return this;});
8  }
9 }
```

Figure 6: Event announcement with event types in C#

Figure 6 illustrates the subject `Point`. Compare it with class point in Figure 1. On line 5, `Point` announces the event `Changed` using the `Changed.Announce` method, similar to event announcement on line 5 of Figure 1. The receiver of the announce method is the event type being announced and the event body is provided as an anonymous lambda statement, lines 6–7. The context variable `fe` is created and set on line 5 by creating the object `Changed.Context`.

```
24 class Enforce {
25  Enforce init(Changed.Register(enforce);}
26  Fig enforce(EventType<Fig, Changed.Context next){
27  Contract.Requires(fe != null);
28  Contract.Ensures(fe != null);
29   if (next.fe.fixed == 0)
30    return next.Invoke();
31   else {
32    Contract.Assert(1==1);
33    return next.context().fe;
34    Contract.Assert(next.Context.fe ==
35        Contract.OldValue(next.Context.fe));
36  }}}
```

Figure 7: Event handler in C#

Figure 7 illustrates the handler method `enforce` on lines 26–36. Compare it with the `enforce` in Figure 1. Event registration is done via the call to the **`register`** method on the event type, line 25. The **`Invoke`** statement is similar to Ptolemy's invoke expression, allowing the next applicable handler to be called. Lines 32-35 are the equivalent of Ptolemy's refining expression on lines 27–28 of Figure 1. Assertion statements on lines 32 and 34–35 are run time probes added to enforce the specification stated by specification expression on lines 20–21 of Figure 5. Ptolemy's quantification mechanism is simulated in C# by the proposed idiom of passing the event type to the handler as a parameter, on line 26 .

4.3 Discussion

As previously mentioned in Section 2, runtime assertions assure that each handler method refines the pre- and post-condition of the event type it handles. They also check that Ptolemy's `refining` expression actually refines the specification it claims. In C# it means the insertion of runtime probes on lines 27–28 of Figure 7 to enforce the contract's pre- and post-conditions, stated on lines 15–16 of Figure 5. Also, the addition of assertions on lines 32 and 34-35 to make sure the specification expression on lines 20-21 of Figure 5 is not violated by any program expression which claims to refine it, line 33 of Figure 7. Insertion of runtime probes and structural refinement of the contract by handlers could be carried out by a simple source to source transformation. The transformation also makes sure that the refining handler methods and each code block constrained by a specification expression have one exit point to avoid unreachable code (line 33, Figure 7) . Structural similarity is crucial to structural refinement [3, 14].

5. RELATED WORK

This work, especially the internals of the translucid contracts, relates to works which propose: (1) behavioral contracts for aspects and (2) modular reasoning techniques for AO interfaces.

Behavioral contracts for Aspects:. Use of behavioral contracts to limit the behavior of aspects for the ease of reasoning is an accepted approach, exercised in the works such as crosscut programming interfaces (XPI) [8, 18], Pipa [19] and Cona [10, 15] among the others. XPI's informal contracts in terms of constraints for the advised and the advising code, Pipa's JML-like annotations and Cona's contracts for both aspects and objects are all behavioral contracts, which makes them incapable of specifying any control effect of interest. Furthermore, there is no verification mechanism proposed for XPI contracts.

Modular Reasoning for AO Interfaces:. Frequent join point shadows are one of the obstacles in modular reasoning about AO programs. Open Modules [1], explicit join points [7], join point types [16] and Ptolemy [13] tackle this problem by limiting the number of join point shadows as we have done in this work. However they do not provide any concrete specification and verification mechanism for reasoning.

Understanding the control effects of the advice is another problem in modular reasoning. "Harmless" advice [5] assumes aspects with no side effects. Categorizing the aspects as assistants (or spectators) [4], which can(not) enhance the behavior of the base code helps with reasoning. EffectiveAdvice [11] proposes explicit advice points and composition and its typed model enforces control and data flow properties. However, its non-AO core makes it difficult to adapt it to II, AO and Ptolemy as it lacks quantification.

6. CONCLUSION

Although implicit invocation (II) improves modularity, it makes modular reasoning difficult especially reasoning about control effects. In the previous work [3] translucid contracts were proposed to enable modular reasoning in Ptolemy. In this work, we show that translucid contracts are independent of their original context, Ptolemy, and are applicable to other AO interfaces. We also propose a simple programming idiom to enable application of translucid contracts to C#. The basic requirement when applying translucid contracts is: for each handler, it should be possible to statically tell which event types it handles. The proposed idiom meets this requirement. The idiom is simple and general and can be applied to other OO languages. Using the idiom makes it possible to know what events a handler method can handle. In summary, translucid contracts are independent of Ptolemy and are applicable to implicit AO and explicit OO event announcement models.

Acknowledgments

Bagherzadeh and Dyer were supported in part by NSF grant CCF-10-17334. The work of Leavens was supported in part by NSF grant CCF-10-17262.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*.

[2] M. Bagherzadeh, H. Rajan, and G. T. Leavens. Translucid contracts for aspect-oriented interfaces. In *FOAL '10*.

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

[4] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about Aspects. In *ECOOP '07*.

[5] D. S. Dantas and D. Walker. Harmless advice. In *POPL'06*.

[6] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. SAC '10.

[7] K. J. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ'07*.

[8] K. J. Sullivan *et al.* Information hiding interfaces for aspect-oriented design. In *ESEC/FSE'05*.

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

[10] D. H. Lorenz and T. Skotiniotis. Extending design by contract for aspect-oriented programming. *CoRR*, abs/cs/0501070, 2005.

[11] B. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: Disciplined advice with explicit effects. In *AOSD'10*.

[12] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *AOSD'6*.

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

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

[15] T. Skotiniotis and D. H. Lorenz. Cona: Aspects for contracts and contracts for aspects. In *OOPSLA'04*.

[16] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1), 2010.

[17] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM*, 20(2), 2009.

[18] W. G. Griswold *et al.* Modular software design with crosscutting interfaces. *IEEE Software'06*.

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