Static Verification of PtolemyRely Programs Using OpenJML

José Sánchez
University of Central Florida
Dept. of EECS, Orlando, FL 32816, USA
sanchez@eecs.ucf.edu

Gary T. Leavens
University of Central Florida
Dept. of EECS, Orlando, FL 32816, USA
leavens@eecs.ucf.edu

Abstract

In the PtolemyRely language *event types* define events that, when announced, trigger the execution of handlers, passing along the triggering piece of code for its eventual execution.

Verification of PtolemyRely programs poses some particular challenges: (1) handlers must be verified against their corresponding event declaration, (2) event announcement and next-handler invocation must be reasoned about according to PtolemyRely's semantics, (3) the body of refining statements must be checked against their specifications, etc. The original Ptolemy compiler includes run-time assertion checking for dynamic verification, but there has been no static verification tool.

In this paper we address the challenge of *static* verification of PtolemyRely programs by encoding them into JML (the Java Modelling Language) and using a JML static verification tool (OpenJML) to discharge the verification obligations. We argue informally that our encoding is *sound* in the sense that a PtolemyRely program is valid if and only if its encoding is a valid JML program.

Categories and Subject Descriptors D.2.1 [*Requirements/Specifications*]: Languages; D.2.4 [*Software/Program Verification*]: Formal Methods, Programming by Contract; F.3.1 [*Specifying and Verifying and Reasoning About Programs*]: Logics of Programs, Pre- And Post-Conditions, Specification Techniques

Keywords Static verification, Ptolemy language, PtolemyRely, OpenJML

1. Introduction

Many approaches have been proposed to deal with the lack of modular reasoning capabilities in aspect-oriented (AO) programs. Most of these approaches rely on some form of interface type that decouples subjects (base code) from handlers (advice). Event types [11], Join Point Types [14] and Joint Point Interfaces (JPI) [2, 3, 7] are some prominent examples.

FOAL '14, April 22 - 26 2014, Lugano, Switzerland.

http://dx.doi.org/10.1145/2588548.2588550

1.1 Event Types in PtolemyRely

An *event type* in PtolemyRely[1] defines an interface between the base code or subject, that announces an event, and the handlers, that are invoked in response. Like Ptolemy, PtolemyRely allows triggering an event from any statement position. Event types also define any context information that is expected by the handlers. **announce** statements trigger events and pass to the handlers the required context information, along with the triggering piece of code (*announced code*) for its eventual execution. The first registered handler, if any, starts execution, otherwise only the announced code is executed. **invoke** statements, in a handler, execute the next handler, if any, or the passed-on announced code, otherwise.

```
1  public class Base {
2    public void run(){
3      Particle p1 = new Particle(1);
4      registerHandler();
5      announce MovingEvent(p1) { // Q_e : x' ≥ 0
6        p1.setX(p1.x()+1);       // Q_r : x' >= x
7      }
8      assert p1.x()>=0; // x' ≥ 0
9   }}
10 public void event MovingEvent {
11   Particle p;
12   relies requires true              // P_r : true
13         ensures p.x()>=old(p.x())// Q_r : x' ≥ x
14   requires true    //P_e : true
15   assumes{
16     next.invoke(); // proceed
17     //spec. expr. P : true Q : x' ≥ 0
18     requires true ensures next.p().x()>=0;
19   }
20   ensures p.x()>=0 //Q_e : x' ≥ 0
21 }
22 public class MovingHandler1 {
23   public void handle(MovingEvent next)
24         throws Throwable{
25    next.invoke(); // proceed
26    refining requires true ensures next.p().x()>=0{
27      if(next.p().x()<0)
28        next.p().setX(1);  // x' = 1
29   }}
30   when MovingEvent do handle;
31   public MovingHandler1(){
32     register(this);
33 } }
```

Figure 1. Particle example in PtolemyRely

[1] PtolemyRely [12] is an extension of the Ptolemy [11] language with separate specifications for the announced code and the handlers.

The example in Figure 1 illustrates the features of PtolemyRely. When a `Particle` changes its position (line 6) it triggers `MovingEvent`, passing `p1` as the context (line 5) and a closure for the announced code (line 6).

`MovingHandler1.handle` (line 23) is bound to the event (line 31). The event itself declares the context (line 11), the specification for any announced code (lines 12-13) and the specification (lines 14, 20) and abstract algorithm (lines 16-18) for any handler.

PtolemyRely also provides specification features. Like Ptolemy, event types in PtolemyRely explicitly address the modular specification and verification problem using *translucid contracts* [1]. The contract establishes the specification (pre, post-conditions) each handler must satisfy and the abstract algorithm they should refine. Specification statements, **`requires`** P **`ensures`** Q, in the abstract algorithm, hide implementations details; **`refining`** statements in the body of each handler provide their particular realizations. PtolemyRely [12] extends Ptolemy by providing separate specifications for the announced code, which are used when reasoning about **`announce`** and **`invoke`** statements. Summarizing, the specification of an event Evt includes: the specification that any announced code must satisfy (P_r, Q_r), the specification for the handlers (P_e, Q_e) and the abstract algorithm (A). This information is given by the $EventSpec$ function:
$EventSpec(Evt) = (P_r, Q_r, P_e, A, Q_e)$.

1.2 JML and OpenJML

JML [4, 8, 9] is a behavioral interface specification language tailored to Java. It allows one to specify the syntactic interface of Java modules (classes or interfaces) and its behavior from the client point of view. JML specifications can be at the module level (class invariants), at the method level (pre- and post-conditions) and at the statement level (assume, assert, etc). Specifications in JML are written using normal Java expressions plus some new notations. For example the notation **`\result`** is used to refer to the result of a method and **`\old()`** refers to the pre-state value of an expression. The `Particle` class example in Figure 2 illustrates some features of JML.

```
1  public class Particle {
2    public int x;
3
4    //@ requires true;
5    //@ assignable this.x;
6    //@ ensures this.x==x;
7    public Particle (int x) {
8      this.x = x;
9    }
10
11   //@ assignable \nothing;
12   //@ ensures \result==x;
13   public int x() {
14     return x;
15   }
16
17   //@ requires true;
18   //@ assignable this.x;
19   //@ ensures this.x==x;
20   public void setX(int x) {
21     this.x = x;
22   }
23 }
```

Figure 2. Particle JML Class

JML also provides advanced features [4] to fully specify and verify object oriented programs. Some of these features deal with frame properties, datagroups, ghost and model fields, specification inheritance, exceptional postconditions, etc.

OpenJML [5] is a set of tools for JML, built by extending the OpenJDK Java tool set. It includes verifying (static checking) and run-time checking of Java programs that are annotated with JML specifications. The aim of OpenJML is to check that the implementation part of the JML program satisfies the given specifications.

OpenJML extends the OpenJDK compiler by parsing the JML specifications, type-checking and statically verifying and/or dynamically checking the implementations against the specifications. It generates a proof script from the JML specifications and the Java code. This proof script is submitted to a SMT prover (like Yices or Z3) for verification or for finding a counterexample.

OpenJML currently checks [6] many aspects of a Java program to verify the conformity with its specifications. A *valid JML program*, so determined by OpenJML, satisfies all these checks. Some of these checks, of particular interest in this work, are detailed in Definition 1.1 (the words in [] are the keywords in the OpenJML documentation [6]).

DEFINITION 1.1. *(OpenJML valid JML program) For a JML program to be valid, as determined by OpenJML, it is required that*

1. *Every method satisfies its specification. Assuming the method precondition ([Precondition]) at the beginning, and using the proof rules of the statements in the method body, the method postcondition holds at the end ([Postcondition]).*
2. *In every method call, its precondition, evaluated with the actual parameter expressions, holds immediately before the call.*
3. *Every explicitly stated assertion holds ([Assert]).*

In this work we formalize an encoding of PtolemyRely's events into JML. Using this encoding we can statically and modularly verify the base code, the event types and the handlers. Each event announcement is verified against the specification of the event, the abstract algorithm of the event is verified, once and for all, against the event specification, and each handler is verified to structurally refine the abstract algorithm, particularly each *refining* statement is verified against its specification.

2. Verification in PtolemyRely

In the framework of partial-correctness Hoare logic, to verify a program means to check that it satisfies its specification. For example, if a statement S is said to be a realization of the specification **`requires`** P **`ensures`** Q, then to verify S means to prove that the Hoare triple $\{P\}S\{Q\}$ holds. That is, if S starts execution in a state where P holds and S terminates normally, then Q holds in the resulting state. Following a compositional technique, the structure of a program's correctness proof mirrors the structure of the program itself, combining the proofs for its statements.

To verify a PtolemyRely program encompasses various specific steps. The body of every handler H should satisfy the event handler's specification, $\{P_e\}\ H\ \{Q_e\}$, and structurally refine the event's abstract algorithm, A, i.e., $A \sqsubseteq H$. Specification statements in that abstract algorithm are refined by **`refining`** statements in the body of the handler. These **`refining`** statements are verified against their specifications. The announced code, B, in every **`announce`** statement, **`announce`** $Evt(\overrightarrow{c})\{B\}$, should satisfy its specification, $\{P_r\}\ B\ \{Q_r\}$, in the corresponding event. **`announce`** and **`invoke`** statements execute either a handler or the announced code, therefore their preconditions, $(P_e \wedge P_r)$, should be verified at each announcement and their postconditions, $(Q_e \vee Q_r)$, should be guaranteed. Verifying the handlers against their specification can be done in a simple indirect way by means of their abstract algorithm, first proving $\{P_e\}\ A\ \{Q_e\}$ and then checking structural

refinement $A \sqsubseteq H$, that guarantees $\{P_e\}\, H\, \{Q_e\}$ using techniques from the work of Shaner et al.[13].

The proof rules for PtolemyRely statements, adapted from [12], are as in Figure 3.

(ANNOUNCE)

$$\frac{(P_r, Q_r, P_e, A, Q_e) = \mathit{eventSpec}(Evt), \quad \vec{x} : \vec{T} = \mathit{formals}(Evt), \quad \Gamma \vdash \{P_r[\vec{y}/\vec{x}]\}B\{Q_r[\vec{y}/\vec{x}]\}}{\Gamma \vdash \begin{array}{c}\{P_e[\vec{y}/\vec{x}] \wedge P_r[\vec{y}/\vec{x}]\} \\ (\textbf{announce}\ Evt(\vec{y})\ B) \\ \{Q_e[\vec{y}/\vec{x}] \vee Q_r[\vec{y}/\vec{x}]\}\end{array}}$$

(INVOKE)

$$\frac{(\mathit{closure}\ Evt) = \Gamma(\textbf{next}), \quad (P_r, Q_r, P_e, A, Q_e) = \mathit{eventSpec}(Evt)}{\Gamma \vdash \{P_e \wedge P_r\}\textbf{next.invoke}\,()\,\{Q_e \vee Q_r\}}$$

(SPECIFICATION-EXPR)

$$\frac{}{\Gamma \vdash \{P\}\textbf{requires}\ P\ \textbf{ensures}\ Q\{Q\}}$$

(REFINING)

$$\frac{\Gamma \vdash \{P\}S\{Q\}}{\Gamma \vdash \{P\}(\textbf{refining requires}\ P\ \textbf{ensures}\ Q\ \{\ S\ \})\{Q\}}$$

Figure 3. Rules for the interesting constructs of PtolemyRely.

Since different techniques can be used to verify each one, separate definitions are given for *structural correctness* and general *correctness* of PtolemyRely programs.

DEFINITION 2.1. *(PtolemyRely structural correctness) A PtolemyRely program is structurally correct iff*

1. *Every handler body, H, structurally refines its corresponding event abstract algorithm, A, i.e., $A \sqsubseteq H$. Each statement in H must exactly match its corresponding statement in A, except for specification statements of the form* **requires** *P* **ensures** *Q in A that must be matched by statements* **refining requires** *P* **ensures** *Q $\{S\}$ in H, where S is the code implementing the specification.*

Structural correctness, as explained in [1], can be implemented as part of the type-checking phase.

DEFINITION 2.2. *(PtolemyRely correctness) A PtolemyRely program is correct iff*

1. *Every event abstract algorithm, A, satisfies that event's handlers specification $\{P_e\}\ A\ \{Q_e\}$.*
2. *(Structural correctness) Every handler body, H structurally refines its corresponding event abstract algorithm, A, i.e. $A \sqsubseteq H$.*
3. *And the program satisfies its specification using the proof rules.*

3. Event Encoding in JML

As already mentioned, JML provides comprehensive specification features for Java programs. Also, its OpenJML implementation supports most of these features, providing not only standard compilation of programs but also static verification and run-time checking of them. Thus our approach is to verify PtolemyRely programs by encoding them as JML programs and using the OpenJML tool capabilities to discharge the verification obligations.

We define translation functions *TR* to translate PtolemyRely constructs into JML constructs. Every event type is translated into a JML class (Java class with JML annotations). The fields in the class are the event context variables. The behavior of any announced code is encoded as a closure with body {**assert** P_r**; assume** $TR_{old}[\![Q_r]\!]$} that fulfils the **relies** specification (P_r, Q_r). The behavior of any handler is encoded as a closure whose expression is the event's abstract algorithm $\overrightarrow{S}$, that must satisfy the event handler's specification (P_e, Q_e).

$TR\,[\![T_e$ **event** Id_e {
$\overrightarrow{f} : \overrightarrow{T}$
relies requires P_r **ensures** Q_r
requires P_e **assumes** $\{\overrightarrow{S}\}$ **ensures** $Q_e\}]\!] =$
class Id_e {
$TR_f[\![\overrightarrow{f} : \overrightarrow{T}]\!]$ *// context variables or fields*
/ * @ **assignable** **nothing**; @ * /
$Id_e()\{\}$ *ctor*
/ * @ **requires** $P_e \wedge P_r$; @ * /
/ * @ **assignable** $\overrightarrow{f}$.*; @ * /
/ * @ **ensures** $TR_{old}[\![Q_e]\!] \vee TR_{old}[\![Q_r]\!]$; @ * /
void invoke(){/ * @ // $\overrightarrow{S}$ *or* (P_r, Q_r)
choose $\{closure(P_e, TR_{old}[\![Q_e]\!], TR_s[\![\overrightarrow{S}]\!])\}$
or $\{closure(P_r, TR_{old}[\![Q_r]\!],$
$\{$**assert** P_r; **assume** $TR_{old}[\![Q_r]\!]\})\}$ @ * / }
}

Invoke and specification statements in the abstract algorithm (handler's closure) require special translation. **Invoke** statements are encoded as calls to the `invoke` method, that nondeterministically executes the handler's closure or the announced-code closure. Every specification statement of form **requires** P **ensures** Q is encoded as the execution of a closure with expression { **assert** $TR[\![P]\!]$; **assume** $TR[\![Q]\!]$} that fulfils the given specification.

$TR_s[\![\overrightarrow{S}]\!] = \overrightarrow{TR_s[\![S]\!]}$ *// statements*

$TR_s[\![$**next.invoke**$\,()\,]\!] =$ **this.**invoke$\,()$ *// invoke*

$TR_s\,[\![$**requires** P **ensures** $Q]\!] =$ *// specification*
$closure(TR_{ex}[\![P]\!], TR_{ex}[\![Q]\!],$
$\{$**assert** $TR_{ex}[\![P]\!]$; **assume** $TR_{ex}[\![Q]\!]\}$

Since all handlers are abstracted by the event contract (specification and abstract algorithm), and that contract is encoded into the handler's closure, every reference to *next* in this closure is translated into a reference to *this*. Also all postconditions are translated in order to map the *old()* construct of PtolemyRely into the *\old()* construct of JML.

$TR_{ex}[\![E]\!] = TR_{old}[\![TR_{next}[\![E]\!]]\!]$ *// expressions*

$TR_{next}[\![$**next**$.E]\!] =$ **this**$.E$ *// next expressions*

$TR_{next}[\![E]\!] = E$ *// otherwise*

$TR_{old}[\![$**old**$(E)]\!] =$ **old**(E) *// old expressions*

$TR_{old}[\![E]\!] = E$ *// otherwise*

Each context variable declaration is translated into a declaration of a field of the class, along with its corresponding accessor method. The accessor method is required since, according to PtolemyRely's syntax, the context should be referenced by those methods (**next**.variable()). The translation of a sequence of context variable declarations is as follows.

$TR_f[\![\overrightarrow{f : T}]\!] = \overrightarrow{TR_f[\![f : T]\!]}$

This translation builds on the translation for individual context variable declarations.

$TR_f[\![f : T]\!] = f : T;$ *// variable declarations*
$/*@$ **assignable** **\nothing**; $@*/$
$/*@$ **ensures** **\result** = **this**.f; $@*/$
$T\ f()\{$**return** **this**.f;$\}$ *// accessor*

Given a block of code S and a pre-post specification (P, Q), the function $closure(P, Q, S)$ returns a block of JML code that, if valid, has the same behavior as S and certifies that S satisfies the given specification, $\{P\}S\{Q\}$. The JML code declares a local closure class with fields F_c corresponding to the closure environment and an `execute` method that corresponds to the closure expression. The fields include any free variable in the block of code, the precondition or the postcondition. The body S_c of the `execute` method corresponds to the block of code S and the method specification (P_c, Q_c) corresponds to the given specification, both with the free variables overwritten by the class fields. If the class is valid in JML it means that S satisfies the given specification. Then the instantiation of the class, the setting of its fields, the execution of the method and the getting of the final values restores the behavior of the original block of code.

$closure\ (P, Q, S) = \{$
class $Clr_s\{$
$\overrightarrow{\text{public } F_c}$; *// free variables:* $free(S, P, Q)$
$/*@$ **requires** P_c; **assignable** F_c; $@*/$
$/*@$ **ensures** Q_c; $@*/$
public **void** $execute()\{S_c\}$ *// closure method*
$\}$
$Clr_s\ clr_s =$ **new** $Clr_s()$; *// instantiation*
$\overrightarrow{clr_s.F_c} = \overrightarrow{free(S, P, Q)}$; *// set*
$clr_s.execute()$; *// execute*
$\overrightarrow{free(S, P, Q)} = \overrightarrow{clr_s.F_c}$; *// get*

In the above, F_c are the fields corresponding to the free variables in S,P or Q, $free = free(S, P, Q) = free(S) \cup free(P) \cup free(Q)$, $P_c = P[free := F_c]$, $Q_c = Q[free := F_c]$ and $S_c = S[free := F_c]$. Figure 4 illustrates the $closure$ function.

```
1  class Clr3 {
2   public Particle p1;
3
4   /*@ requires p1 != null; assignable \nothing; @*/
5   /*@ ensures \result == p1; @*/
6   public Particle p1() { return p1; }
7
8   //@ assignable \nothing;
9   public Clr3() { super(); }
10
11  /*@ requires true; assignable p1.x; @*/
12  /*@ ensures p1.x() >= \old(p1.x()); @*/
13  public void execute() { p1.setX(p1.x() + 1);}
14 }
15 Clr3 clr3 = new Clr3();
16 clr3.p1 = p1;
17 clr3.execute();
18 p1 = clr3.p1;
```

Figure 4. Closure of announced code (figure 1, line 6)

An **announce** statement in the base code triggers the execution of the first handler in the chain of handlers, if there are any, or executes the announced code otherwise. The behavior of any handler for the announced event is encoded as a closure with body: $\{$**assert** $P_e[\overrightarrow{f} := \overrightarrow{c}]$; **assume** $TR_{old}[\![Q_e[\overrightarrow{f} := \overrightarrow{c}]]\!]\}$ that fulfils the handlers specification (P_e, Q_e) for that event, with the event's context variables $\overrightarrow{f}$ overwritten by the actual context expressions $\overrightarrow{c}$. The behavior of the announced code is encoded as a closure whose body is that announced code, that must satisfy the corresponding specification in the event declaration (P_r, Q_r). The **announce** statement itself is encoded as a Jml **choose** construct, that non-deterministically executes the handler's closure or the announced code closure.

If $eventSpec(Id_e) = (P_r, Q_r, P_e, \overrightarrow{S}, Q_e)$ and
$formals(Id_e) = \overrightarrow{f} : \overrightarrow{T}$ then
$TR_s\ [\![$**announce** $Id_e(\overrightarrow{c})\{B\}]\!] = /*@$
choose $\{closure(P_e[\overrightarrow{f} := \overrightarrow{c}], TR_{old}[\![Q_e[\overrightarrow{f} := \overrightarrow{c}]]\!],$
$\{$**assert** $P_e[\overrightarrow{f} := \overrightarrow{c}]$; **assume** $TR_{old}[\![Q_e[\overrightarrow{f} := \overrightarrow{c}]]\!]\}\}$
or $\{closure(P_r[\overrightarrow{f} := \overrightarrow{c}], TR_{old}[\![Q_r[\overrightarrow{f} := \overrightarrow{c}]]\!],$
$TR_s[\![B]\!])\}$ $@*/$

Handlers in PtolemyRely are methods in their classes. The type checking phase of the Ptolemy compiler checks that each handler method only has one parameter of the type corresponding to the event closure environment. In this phase it is also checked that the body of the handler method structurally refines the event's abstract algorithm. The verification phase only needs to verify the **refining** statements in the method body. Each **refining** statement **refining** **requires** P **ensures** Q is encoded as the execution of a closure with body S, that must satisfy the given specification.

$TR\ [\![$**refining** **requires** P **ensures** $Q\ \{S\}]\!] =$
$closure(TR[\![P]\!], TR[\![Q]\!], TR[\![S]\!])$

4. Soundness

The *TR* functions translate an original structurally-valid PtolemyRely program into an encoded JML version. The encoded version can be checked by the extended static checker (*esc*) of OpenJML. The soundness of the translation *TR* requires that whenever the encoded version is proved correct by OpenJML, as stated in Definition 1.1, then the original version is a correct PtolemyRely program, satisfying the conditions in Definition 2.2.

We now present an informal argument of the soundness of the encoding; for reasons of space we do not present here the complete proof. The following observations support the soundness argument:

- A block of code, S, satisfies a specification iff its closure also satisfies it: $\{P\}S\{Q\} \Leftrightarrow \{P\}closure(P, Q, S)\{Q\}$. That follows from the structure of the code produced by the *closure* function. It encapsulates the code S as the body of a method, with the corresponding specification **requires** P **ensures** Q, so the code satisfies the specification iff the method is valid in JML. The invocation of the method restores the effect of the code in its original context.
- A block of code in a PtolemyRely program satisfies a specification iff its translation satisfies that specification: $\Gamma_{PR} \vdash \{P\}S\{Q\} \Leftrightarrow \Gamma_{JML} \vdash \{P\}TR[\![S]\!]\{Q\}$. That follows from the previous observation about the *closure* function and from the encoding of most of the PtolemyRely interesting constructs using that *closure* function and the JML **choose** construct.
- Finally, the soundness: a structurally-correct PtolemyRely program P is correct iff its translation $TR[\![P]\!]$ is a valid Jml program. Each event is encoded as a class. The abstract algorithm becomes the body of a method with the event handlers specification. Again, the abstract algorithm will satisfy the handlers specification iff the method is valid in JML.

5. Related Work

The original design of Ptolemy [11], and others similar approaches like Join Point Types [14] and Join Point Interfaces (JPI) [2, 3, 7], address the problem of modular reasoning of aspect oriented systems by providing some sort of interface type that decouples subjects from handlers.

The use of *translucid contracts* [1] was introduced as a way to specify and verify the *event types* that lie at the core of Ptolemy language. That work also enhanced the Ptolemy compiler with a Run-Time Assertion Checker (RAC) that implements the verification of Ptolemy programs specified by translucid contracts.

In a previous work [12] the authors extended the specification and verification capabilities of Ptolemy by explicitly separating the specification for the handlers from the specification for the announced code, making the approach more flexible and complete.

There has been a prolific amount of work on JML since its conception [4, 8–10] and on tools for verifying JML programs, like OpenJML [5].

6. Conclusions and Future Work

Event types are used in PtolemyRely to decouple subjects, which announce events, from handlers. The expected behavior of both the handlers and the announced code are specified as part of the event definition using translucid contracts.

In this paper we formalized an encoding of PtolemyRely's events into Java Modelling Language (JML). Then we can use a JML tool, like OpenJML, to statically verify the encoded program. Our encoding is sound in the sense that a PtolemyRely program is valid if and only if its encoding is a valid JML program.

We are currently working on a prototype implementation of a PtolemyRely static verifier. It takes a PtolemyRely program, translate it into a JML version, encoding the PtolemyRely obligations into JML constructs, and then uses the OpenJML extended static checker (esc) to validate the JML version and hence the original PtolemyRely program.

Taking advantage of the modular design of OpenJML (that extends OpenJDK 7) our implementation also follows a modular design that extends OpenJML.

Since some features of JML are not currently supported by OpenJML, we used other less suitable but still useful constructs. A future work that can be done is to implement some of the missing features in OpenJML, that would allow a simpler encoding of PtolemyRely.

Since mechanisms have been recently proposed for the verification of event subtyping in Ptolemy our translation could be extended to also encode and verify event sybtyping.

Acknowledgments

The work of both authors was partially supported by NSF grant CCF-1017334. The work of José Sánchez is also supported by Costa Rica's Universidad Nacional (UNA), Ministerio de Ciencia y Tecnología (MICIT) and Consejo Nacional para Investigaciones Científicas y Tecnológicas (CONICIT).

References

[1] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. 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. ACM, 2011.

[2] E. Bodden. Closure joinpoints: Block joinpoints without surprises. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 117–128. ACM, 2011.

[3] E. Bodden, E. Tanter, and M. 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.

[4] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/-Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, Berlin, 2006. Springer-Verlag. URL `http://dx.doi.org/10.1007/11804192_16`.

[5] D. Cok. OpenJML: JML for Java 7 by extending OpenJDK. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer-Verlag, Berlin, 2011. . URL `http://dx.doi.org/10.1007/978-3-642-20398-5_35`.

[6] D. Cok. Openjml checks, 2014. URL `http://jmlspecs.sourceforge.net/checks.shtml`.

[7] M. Inostroza, E. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 508–511, 2011. URL `http://www.bodden.de/pubs/itb11jpi.pdf`.

[8] G. T. Leavens and Y. Cheon. Design by contract with JML. Draft, available from jmlspecs.org., 2005. URL `http://www.jmlspecs.org/jmldbc.pdf`.

[9] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006. URL `http://doi.acm.org/10.1145/1127878.1127884`.

[10] G. T. Leavens, P. H. Schmitt, and J. Yi. The Java Modeling Language. NII Shonan meeting report. Technical Report 2013-3, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan, May 2013.

[11] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In J. 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. URL `http://dx.doi.org/10.1007/978-3-540-70592-5_8`.

[12] J. Sánchez and G. Leavens. Separating obligations of subjects and handlers for more flexible event type verification. In *SC 2013: Proceedings of the 12th International Conference*, pages 65–80. Lecture Notes in Computer Science, June 2013. URL `http://dx.doi.org/10.1007/978-3-642-39614-4_5`.

[13] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367, New York, NY, Oct. 2007. ACM. . URL `http://doi.acm.org/10.1145/1297027.1297053`.

[14] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, pages 1:1–1:43, 2010.