Combining Event-driven and Capsule-oriented Programming to Improve Integrated System Design

Jackson Maddox[*]

Iowa State University, Ames, IA, 50011, USA,
jlmaddox@iastate.edu

Abstract. As concurrent software becomes more pervasive, models that provide both safe concurrency and modular reasoning become more important. Panini is one such model, and provides both sparse and cognizant interference based around the concept of capsules. Additionally, web frameworks, Graphical User Interface (GUI) libraries, and other projects are event-driven in nature, making events a commonly used programming paradigm for certain tasks. However, it would be difficult to use Panini in an event-driven manner, where there may be multiple capsules interested in a given event. Therefore, by integrating capsules and events one would be able to apply Panini's modular reasoning to commonly event-driven tasks more easily. Several challenges must be addressed in the integration. These are defining the semantics of event messages, scheduling of handlers to maximize concurrency, and how to keep to Panini's current semantics which allow modular reasoning. To solve this problem, @Paninij, an implementation of Panini, is extended to add event mechanisms to capsules. As a result, this new combined model allows capsules to interact using both procedures and event announcements. This extension of Panini is helpful for writing concurrent, modular software that lends itself more naturally to event-driven programming.

1 Introduction

Many modern software projects are both concurrent and event-based. Unfortunately, programming concurrent software is prone to subtle concurrency bugs and can be very difficult to reason about. The notion of capsule-oriented programming [1] in the Panini project [2–4] provides safe and modular concurrent programming. These capsules act as modules of independent execution in a software, and interact with each other by thread-safe message passing. They also provide a model under which a programmer can more easily reason about their concurrent programs. Over the past decade, much work has gone into advancing Panini's notion of capsule-oriented programming. For example, some works have advanced the language design [1, 5–7], the semantics [8], compilation strategy [9], and type-and-effect systems [10, 11]. However, Panini's capsules do not provide features for handling more dynamic event-based systems.

This paper describes an extension of Panini to provide mechanisms for event-driven communication between capsules. Event-driven programming, also called implicit invocation [12, 13], has been shown to be highly effective for component integration and

[*] This report describes a project conducted as part of Iowa State University Honors program under the guidance of Dr. Hridesh Rajan.

reuse [14, 15] and over the last couple of decades has been combined with a number of programming paradigms, e.g. with aspect-oriented programming [16–20]. However, reasoning about event-driven programming is also shown to be challenging [21–23] and solutions have been proposed to improve reasoning about events, e.g. the design of the Ptolemy language was motivated by the need to improve reasoning about events [24–26], the idea of *translucid contracts* proposed specifications techniques to ease reasoning about events [27–29], and mixing events and subtyping was shown to be a challenge [30, 31]. A key challenge for combining events and capsules is to maintain the reasoning properties of capsules as described by the prior work [8].

In Panini, capsules and their relationships are defined statically at compile-time, and they cannot be passed around or created once the program has begun execution. This makes it difficult to implement a dynamic event-driven program using Panini's capsule constructs since we would not be able to register an arbitrary number of capsules on a particular event. Since many modern programs, especially ones who have some sort of graphical user interface (GUI), are extremely event-driven in nature, the current Panini model may therefore be difficult to use. In the case of a user interface, there is normally at least two modules: the GUI module, and a module for the logic, each in their own thread. A capsule for the GUI, in absence of events, would be required to know all the capsules interested in each event, and each logic capsule will need to be able to update the GUI, tightly coupling the two. Providing capsules with an event mechanism overcomes the problem by allowing capsules to define events without knowing which capsules have subscribed.

Our goal is to keep the current semantics and guarantees that Panini makes for reasoning about concurrent programs as much as possible. To that end, capsules should generally stay as static entities after a program begins execution and all interactions between capsules should still be thread-safe and provide the same ability to modularly reason about programs. Since we want events from one thread to trigger some logic on another thread, we therefore add the event mechanism on the capsules directly. This mechanism allows capsules to define events, handlers for events, and explicitly announce these events to subscribed capsules. In this way event announcements are similar to invoking capsule procedures except on an arbitrary set of interested capsules.

The @Paninij project [32] is an implementation of Panini's capsule-oriented programming model as a Java annotation processor. In order to demonstrate the capsule event-driven mechanisms, @Paninij was extended with the new event-driven features. Multiple questions needed to be answered in order to complete the design and implementation of these new features. Firstly, how are events to be announced, and what happens to the ownership of the object that acts as the announcement message? It is possible that this message will be sent to multiple capsules who have subscribed to an event. Secondly, in what way are subscribed handlers scheduled to run once an event has been announced?

This new event mechanism for capsules allows programmers to define events that are intended to be announced by different threads or modules of a program. It allows Panini to be used to more easily develop concurrent software with event-driven components such as GUIs.

The structure of the following paper is to first elaborate on the relevant parts of the @Paninij project, discuss the design of the event-driven capsule mechanisms, and finally discuss the implementation of these mechanisms.

2 @Paninij

The @Paninj project implements the Panini programming model as a Java Annotation Processor. At a high level, it provides a way for developers to define a capsule as a class and its procedures as methods of that class. The class the developer provides is called a **Capsule Core** and provides the capsule's logic. Then, using this core, @Paninij creates a new class called the **Capsule**, which handles all thread and synchronization logic. This capsule then uses the core to implement the capsule's procedure logic. As the developer writes their code, they reference these generated capsules to invoke other capsules' procedures. Like procedures in Panini, calling a method on a capsule object adds a message to the end of the capsule's queue for processing and can also either blocking or operate asynchronously. Once these capsules have been generated, they can be used to begin executing a capsule system. Below we will discuss the relevant portions of @Paninij to implementing event mechanisms. **Listing 1.1.** shows a simplified @Paninij program that echoes text read from the console back to the user.

Listing 1.1. A simplified @Paninij program

```java
@Root @Capsule class EchoCore {
  @Local StdInput i;
  @Local StdOutput o;
  void design(Echo self) { i.imports(o); }
  public static void main(String[] args) {
    CapsuleSystem.start(Echo.class, args);
  }
}
@Capsule class StdOutputCore {
  public void echo(String s) {
    System.out.println(s);
  }
}
@Capsule class StdInputCore {
  @Imported StdOutput o;
  Scanner scan = new Scanner(System.in);
  void run() { while (true) o.echo(scan.next()); }
}
```

2.1 Defining a capsule system

In order to define a capsule in @Paninij, the programmer first creates a class with the desired name and appends "Core" to that name. The programmer then decorates the class declaration with `@Capsule`. This tells the annotation processor to generate a capsule of that class's name without the "Core" suffix that uses the programmer defined class as a concrete implementation of the capsule's instance variables and procedures. There are two types of capsules that can be defined: active and passive. Active capsules have no procedures but may have a `run()` method that is executed after the capsule finishes initialization, and passive capsules have procedures but no `run()` method. Once a set of capsules have been defined, there must be an entry point to the capsule system called a Root, denoted by adding the `@Root` annotation alongside `@Capsule`. This Root capsule must be active, and its `run()` method acts as an entry point to the capsule system's execution. Finally, the developer can begin execution of the capsule system calling the static `CapsuleSystem.start(root, args)` method. This method instantiates the capsule system, wires capsules together, and begins executing active capsules.

2.2 Defining capsule procedures

In Panini, capsules can interact with each other in a thread-safe manner via procedures. These procedures and their respective procedure calls have the same syntax as methods and method calls in Java. Since @Paninij's capsules are designed by creating a capsule Core class, similarly procedures can be defined as public methods in the Core class. Then the processor generates the equivalent method on the generated capsule class. When calling the generated method, a message is placed at the end of the capsule's execution queue. The capsule will then process this message using the equivalent Core class's method and send a message with the result, if any, back to the calling capsule. Depending on the way the method is defined, and the annotation decorating it, the method can either act as a blocking method or an asynchronous method that can return either a future or a duck future of the return type when called by other capsules.

2.3 Capsule wiring

In order to declare a relationship between two capsules, the programmer creates an instance variable of a capsule type. This variable is then annotated with `@Local` if the capsule needs its own instance or `@Imported` if it wants the instance to be provided by another capsule. For capsule fields annotated with `@Local`, the capsule will be instantiated automatically when the capsule is constructed. If a capsule has an `@Imported` field, then any owning capsule must implement the special `design(CapsuleName self)` method. This method is similar to the `design` construct in Panini, used exclusively to wire capsules together, in this case via the automatically generated method `capsule.imports(capsuleToImport)`. Finally, there is one other special method to mention: `init()`. This method is called once after a capsule has been fully wired.

3 Event Design and Semantics

This section describes the design and semantics of each modification to the Panini model. Notice that Panini's core semantics stay the same, and the event mechanism acts as an add-on to capsules themselves. The main challenge in designing this integrated model is deciding how to maximize the usability and concurrency of the event mechanism while also preserving Panini's semantics. These semantics are particularly important to preserve since they allow for modularly reasoning about a given capsule system.

In **Listing 1.2**. the program shown from the previous section was modified to use capsule events. Clearly this example is a bit more complicated than the first. However, consider adding a capsule to this system that outputs the user input to another location such as a file or across the network. In order to do this, we would need to write the new capsule's core and modify `EchoCore` to wire the new capsule up.

Now consider the example from the previous section. In that example we would also need to modify `StdInputCore` to add another `@Imported` field and change its `run()` method to notify two output capsules rather than just one. However, no such modifications are needed to `StdInputCore` when using events since it does not know about the output capsules. Instead, it announces its event to any capsules interested in listening.

Listing 1.2. Echo program with events

```
@Root @Capsule class EchoCore {
  @Local StdInput i;
  @Local StdOutput o;
  void design(Echo self) { o.imports(i); }
  public static void main(String[] args) {
    CapsuleSystem.start(Echo.class, args);
  }
}
@Capsule class StdOutputCore {
  @Imported StdInput i;
  void design(StdOutput self) {
    i.onInput().register(self::echo, RegisterType.READ);
  }
  @Handler public void echo(String s) {
    System.out.println(s);
  }
}
@Capsule class StdInputCore {
  Event<String> onInput;
  Scanner scan = new Scanner(System.in);
  void run() { while (true) onInput.announce(scan.next()); }
}
```

3.1 Events

Events are defined as instance variables in the capsule core. They are of the type `Event<T>` where `T` is the type of message that will be sent to registered event handlers from other capsules upon announcement. Event instances, like `@Local` capsule instance variables, are automatically instantiated when the capsule is initialized. When an event is defined, a method with the same name as the event variable will be created in the generated capsule. This method takes no arguments and returns a type of `Event<T>`. Its only purpose is to allow other capsules to retrieve a reference to the event object in order to register their handlers as discussed later. With that reference, another capsule can register their event handlers.

3.2 Event Handlers

An event handler in @Paninij is a special type of procedure denoted with the `@Handler` annotation. Like capsule procedures, it is defined as a public method on the capsule Core. However, the handler annotation places further restrictions on the handler's definition. Event handlers must return void and take a single argument of type `T` which will be the message announced by the events that the handler is registered to. If the handler has been registered to a capsule's event, it will eventually be called equivalently to a normal capsule procedure. That is, it will execute on the thread of the capsule defining it.

However, there is one important semantic difference between capsule procedures and event handlers. Like capsule procedures, invoked handlers will place a message on the capsule's queue and execute on the capsule's thread. Also like capsule procedures, event handlers have ownership of its parameters. However, the capsules does not retain ownership of the parameter afterwards. Instead, once a handler has finished executing, it loses ownership of the event message so that ownership can be transferred to any remaining capsule handlers and then back to the announcing capsule. This semantic difference is equivalent to a capsule procedure who always returns its parameter.

3.3 Registration

Since a capsule system's hierarchy is rigid once the system has started, it makes sense to follow this design and make handler registration rigid as well. However, capsules may also lose or gain interest in events over a program's execution, so we must also support some measure of dynamic registration.

Registration is done in the `design()` method of a capsule, alongside the capsule wiring step. In order for a capsule to register a handler to another capsule `c`'s event, it must first have a reference to `c` through either a `@Local` or `@Imported` instance variable. Recall that the `design()` method requires a single argument to receive a reference to the capsule's actual instance. The programmer then retrieves the event from the capsule and calls its `register()` method, providing a method reference to the generated handler from the `design()` method's self parameter and a `RegisterType` enum value.

This `RegisterType` has two values: `READ` and `WRITE`. `READ` may be used only when the handler will never modify the event message whereas `WRITE` should be used when it might modify it. The difference between a `READ` and `WRITE` handler is necessary, as discussed in the next section, to maximize concurrency when invoking capsule handlers. Therefore, when registering an event handler, the developer must know what kinds of effects the handler may have on the event message. They then register each handler appropriately as either `READ` or `WRITE`.

One further aspect of registering handlers, is that while they must be defined at the beginning of program execution in the `design()` method, they can be toggled. Specifically, the `register()` method returns an object of type `EventConnection<T>` with two methods `on()` and `off()`. These methods, when called, will respectively enable or disable the handler's registration. However, disabling will only take effect on future event announcements, and not in-progress announcements. This feature gives some measure of control to the registered capsule and effectively allows it to unregister from an event at will.

3.4 Announcement

Only the capsule that owns an event should ever announce it. Event announcements are explicit: they are initiated by calling `e.announce(T msg)`, providing the event message to transfer to handling capsules. This method also returns an object of type `EventExecution<T>`, which allows an announcer to either block until all interested handlers have completed, or to check whether that has occurred. Recall that the semantics of handlers is specified to release ownership of the message once it has completed execution. This therefore allows the event announcer to know when it gets back ownership of the announced message. This can be helpful in the case of `WRITE` handlers which may modify the message to change the outcome of an event.

The execution of a particular event announcement forms a chain, ordered by the time a capsule registered to the event. In essence, this means that handler registrations within a single capsule will run in the order they are registered. However, there may be interleaving handler executions in between. The reason for this nondeterminism is because in @Paninij, capsules are initialized and then run their `design()` method on their capsule thread. As each capsule executes `design()` on their own thread, the threads of other capsules may begin running either before or after its own. However, we preserve the order of handler invocations within a single capsule because it is strikingly similar to calling capsule procedures. This keeps to Panini's semantics in which the order of procedure invocations not lost, though other capsules may also invoke procedures in-between.

Each link of the event execution chain, however, is a set of handlers rather than a single handler. To be more specific, each link in the chain contains either one or more `READ` registrations **or** a single `WRITE` registration. This allows sets of `READ` handlers to be invoked at the same time whereas each `WRITE` handler will have exclusive access to the event message. Using the strategy we can therefore keep to Panini's semantics while maximizing the concurrency of an event announcement.

At the same time, some messages may be immutable, or the announcing capsule does not wish handlers to modify the object. In such cases an event definition can be annotated with `@Broadcast`, which forces all handlers to register as `READ`. With `@Broadcast`, therefore, the event message is sent to all registered handlers at the same time, following the registration order. Another annotation, `@Chain`, is the default event type which allows both `READ` and `WRITE` handlers to register.

4 Implementation

The @Paninij project has two main parts: the annotation processor for generating capsule classes from cores, and the runtime which the capsules depend on. This section will discuss how these two parts implement events, handlers, registration, and announcements. When implementing the integrated model in @Paninij, the main concern that arose was deciding how to correctly invoke each registered handler according to the scheduling strategy discussed in **Section 3.4**. The code in this section will generally show only interfaces of the new objects added to the runtime for supporting this event mechanism. Synchronization concerns will assume to be handled in the implementation.

4.1 Events and registration

In order to support events, two annotations were added to the runtime: `@Broadcast` and `@Chain` which defines the event behavior as previously discussed. Additionally, an enum `RegisterType` with values `READ` and `WRITE` were created to state whether or not a handler will mutate an event's message when announced. Internally, another enum `EventMode` with values `CHAIN` and `BROADCAST` is used to specify how an event will behave when announced. Finally, two classes were also added to the runtime.

The first of these is `Event<T>` whose interface is shown in **Listing 1.3.** This class provides a method that allows other capsules to register their handlers, and a method that allows a capsule to announce an event. The other class is `EventConnection<T>`, shown in **Listing 1.4.** This class represents a registered handler and its type. Event connections can also be enabled or disabled at any time, changed only by the capsule who registered.

Listing 1.3. The Event<T> class

```java
public class Event<T> {
    Event(EventMode mode);

    /* register the given handler to this event */
    public EventConnection<T> register(
        BiConsumer<EventExecution<T>, T> handler,
        RegisterType type);

    /* announce an event to all enabled registered capsules */
    public EventExecution<T> announce(T message);
}
```

Listing 1.4. The EventConnection<T> class

```java
public class EventConnection<T> {
    EventConnection(
        BiConsumer<EventExecution<T>, T> handler,
        RegisterType type);

    /* enable the handler to receive new announcements */
    public void on();

    /* stop the handler from receiving new announcements */
    public void off();
}
```

When an event is defined in a capsule, it is internally, like with `@Local` capsules, instantiated for the core in the capsule's constructor. At construction, the `Event<T>` is provided with an `EventMode` value to assign its announcement behavior. Since this is done in the capsule constructor, it is always available to the core to use in procedures and other capsules to register on.

To support event registration, a method is generated on the capsule with the same name that takes no arguments and simply provides the event instance. Then once a capsule registers to the event, a new `EventConnection<T>` is created, stored in the `Event<T>`, and returned to the registering capsule. The `Event<T>` object will later use this to begin an event announcement.

4.2 Handlers and announcements

In order to implement handlers, there must be a way to describe them. Enter the `@Handler` annotation which notifies the annotation processor to generate an event handler rather than a capsule procedure. First note that most procedure signatures are altered slightly from the core's method signature. For instance, a procedure denoted to return a future for a type `T`, will have the core method returning type `T` whereas the capsule method will return type `Future<T>`. A similar thing happens to event handlers, except the return type is not altered from `void`. Instead, if a handler's signature in the capsule core is `@Handler public void handler(T msg)`, the generated signature will be `public void handler(EventExecution<T> ex, T msg)`.

This new class, `EventExecution<T>` (**Listing 1.5.**), handles executing a chain of event handlers with a particular message. When an event is announced, a new instance of `EventExecution<T>` is created with a reference to the announced message and all registered and enabled `EventConnection<T>` objects. This instance is then instructed to begin executing handlers and is then returned to the announcing capsule. Using this method, we enable the announcing capsule to wait for an announcement to complete and to reclaim ownership of the event message.

Listing 1.5. The EventExecution<T> class

```
public class EventExecution<T> {
    EventExecution(List<EventConnection<T>> handlers);

    /* have all handlers completed? */
    public boolean isDone();

    /* block until all handlers have completed */
    public void done()
    /* called by Event<T> to begin invoking handlers */
    void execute(EventMode mode, T arg);

    /* called when a handler completes */
    public void panini$markComplete();
}
```

The final missing piece is how these handlers and the executor interact. The executor, when invoked, will begin calling event handlers using the strategy defined by the registration type and event mode as described in **Section 3.** However, it must have some way to know when a handler has completed, which is why handlers have a special annotation. When the executor invokes an event handler, it passes itself as one of the arguments. The capsule's handler method will then package the executor and event message into a single object and place it on the capsule queue. When the capsule dequeues the object, it does the following:

1. Extract the event message
2. Call the core's handler method with the event message
3. Extract the executor and call `panini$markComplete`

Once the `panini$markComplete` method has been called, it knows that an event handler has completed. From there, it either waits for the rest of the handlers to complete in the case of several `READ` handlers being invoked, or immediately invokes the next handler.

5 Conclusion

In Panini, it is difficult to model event-driven tasks such as separating a UI and its logic by using only capsules and procedure invocations. This paper described a model combining both capsules and events, which improves Panini's usability for event-driven tasks. To develop this combined model, Panini's capsule model was extended with an event mechanism so that capsules could communicate not just by procedure invocations, but also by event announcements. This resulting model was then implemented in @Paninij, a Java annotation processor implementation of the Panini language.

6 Future Work

In order to prevent data races handlers must both have exclusive access when modifying the message and give up ownership after it has completed. Additionally, the safety of the current handler invocation strategy depends on the programmer correctly classifying event handlers' effects on event messages as READ or WRITE. Since this work extended @Paninij, a Java annotation processor, the program AST is unavailable for analysis. In addition there are related concerns such as dynamic dispatch that make verification difficult to implement precisely. To solve these problems, future work may develop a better strategy for verifying ownership and event handler effects, and identify appropriate tradeoffs.

References

1. Rajan, H.: Capsule-oriented programming. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. ICSE '15, Piscataway, NJ, USA, IEEE Press (2015) 611–614
2. Rajan, H.: Building scalable software systems in the multicore era. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. FoSER '10, New York, NY, USA, ACM (2010) 293–298
3. Rajan, H., Kautz, S.M., Rowcliffe, W.: Concurrency by modularity: Design patterns, a case in point. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '10, New York, NY, USA, ACM (2010) 790–805
4. Long, Y., Mooney, S.L., Sondag, T., Rajan, H.: Implicit invocation meets safe, implicit concurrency. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. GPCE '10, New York, NY, USA, ACM (2010) 63–72
5. Rajan, H., Kautz, S.M., Lin, E., Mooney, S.L., Long, Y., Upadhyaya, G.: Capsule-oriented programming in the Panini language. Technical Report 14-08, Iowa State University (2014)
6. Rajan, H., Kautz, S.M., Lin, E., Mooney, S.L., Long, Y., Upadhyaya, G.: Capsule-oriented programming. Technical Report 13-01, Iowa State University (2013)
7. Rajan, H., Kautz, S.M., Lin, E., Kabala, S., Upadhyaya, G., Long, Y., Fernando, R., Szakács, L.: Capsule-oriented programming. Technical Report 13-01, Iowa State University (2013)
8. Bagherzadeh, M., Rajan, H.: Panini: A concurrent programming model for solving pervasive and oblivious interference. In: Proceedings of the 14th International Conference on Modularity. MODULARITY 2015, New York, NY, USA, ACM (2015) 93–108
9. Upadhyaya, G., Rajan, H.: Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2015, New York, NY, USA, ACM (2015) 840–859
10. Long, Y., Bagherzadeh, M., Lin, E., Upadhyaya, G., Rajan, H.: On ordering problems in message passing software. In: Proceedings of the 15th International Conference on Modularity. MODULARITY 2016, New York, NY, USA, ACM (2016) 54–65
11. Long, Y., Rajan, H.: A type-and-effect system for asynchronous, typed events. In: Proceedings of the 15th International Conference on Modularity. MODULARITY 2016, New York, NY, USA, ACM (2016) 42–53
12. Sullivan, K.J., Notkin, D.: Reconciling environment integration and component independence. SIGSOFT Software Engineering Notes **15**(6) (December 1990) 22–33
13. Garlan, D., Notkin, D.: Formalizing design spaces: Implicit invocation mechanisms. In: VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I, London, UK, Springer-Verlag (1991) 31–44
14. Sullivan, K.J., Kalet, I.J., Notkin, D.: Evaluating the mediator method: Prism as a case study. IEEE Transactions on Software Engineering **22**(8) (August 1996) 563–579
15. Rajan, H.: One more step in the direction of modularized integration concerns. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 36–38
16. Rajan, H., Sullivan, K.J.: Unifying aspect- and object-oriented design. ACM Trans. Softw. Eng. Methodol. **19**(1) (August 2009) 3:1–3:41
17. Rajan, H.: Design pattern implementations in eos. In: Proceedings of the 14th Conference on Pattern Languages of Programs. PLOP '07, New York, NY, USA, ACM (2007) 9:1–9:11
18. Rajan, H.: Unifying Aspect- and Object-Oriented Program Design. PhD thesis, The University of Virginia, Charlottesville, Virginia (August 2005)

19. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: the international conference on Software engineering (ICSE). (2005) 59–68
20. Rajan, H., Sullivan, K.: Eos: instance-level aspects for integrated system design. In: the European software engineering conference and international symposium on Foundations of software engineering (ESEC/FSE). (2003) 297–306
21. Dingel, J., Garlan, D., Jha, S., Notkin, D.: Reasoning about implicit invocation. SIGSOFT Software Engineering Notes **23**(6) (November 1998) 209–21
22. Xu, J., Rajan, H., Sullivan, K.: Aspect reasoning by reduction to implicit invocation. In Clifton, C., Lämmel, R., Leavens, G.T., eds.: FOAL: Foundations Of Aspect-Oriented Languages. (March 2004) 31–36
23. Xu, J., Rajan, H., Sullivan, K.J.: Understanding aspects via implicit invocation. In: 19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria, IEEE Computer Society (2004) 332–335
24. Rajan, H., Leavens, G.T.: Quantified, typed events for improved separation of concerns. Technical Report 07-14d, Iowa State University (2007)
25. Rajan, H., Leavens, G.T.: Design, semantics and implementation of the Ptolemy programming language: A language with quantified typed events. Technical Report 15-10, Iowa State University (2015)
26. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Proceedings of the 22nd European Conference on Object-Oriented Programming. ECOOP '08, Berlin, Heidelberg, Springer-Verlag (2008) 155–179
27. Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: 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, New York, NY, USA, ACM (2011) 141–152
28. Fernando, R.D., Dyer, R., Rajan, H.: Event type polymorphism. In: Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages. FOAL '12, New York, NY, USA, ACM (2012) 33–38
29. Bagherzadeh, M., Rajan, H., Darvish, A.: On exceptions, events and observer chains. In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development. AOSD '13, New York, NY, USA, ACM (2013) 185–196
30. Bagherzadeh, M., Dyer, R., Fernando, R.D., Sánchez, J., Rajan, H.: Modular reasoning in the presence of event subtyping. In: Proceedings of the 14th International Conference on Modularity. MODULARITY 2015, New York, NY, USA, ACM (2015) 117–132
31. Bagherzadeh, M., Dyer, R., Fernando, R.D., Sánchez, J., Rajan, H.: Modular reasoning in the presence of event subtyping. In: Transactions on Modularity and Composition, special edition: Best papers of Modularity'15. (2016)
32. Johnston, D., Mills, D., Erenberger, T., Maddox, J., Rajan, H.: @PaniniJ: An Annotation-based Realization of Capsule-oriented Programming. `https://paninij.github.io/` Accessed: 2017-06-28.