Reconciling Concurrency and Modularity with Pāñini's Asynchronous Typed Events

Yuheng Long Hridesh Rajan Sean L. Mooney
Dept. of Computer Science, Iowa State University
{csgzlong,hridesh,smooney@iastate.edu}

Abstract

This poster presents our language design called Pāñini. It focuses on Pāñini's asynchronous, typed event which reconciles the modularity goal promoted by the implicit invocation design style with the scalability goal of exposing concurrency between the execution of subjects and observers.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Concurrent, distributed, and parallel languages; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures, Patterns

General Terms Languages, Design, Performance

Keywords Safe Implicit Concurrency, Modularity

1. Introduction

We present our implicitly concurrent language Pāñini [3]. The overall goal of its design is to discourage the use of explicitly concurrent features such as threads and locks. Using explicitly concurrent features can complicate program design, development, verification, and maintenance. Instead Pāñini's design encourages programmers to construct their system to improve modularity in its design. If programmers use Pāñini's features to improve modularity, they automatically receive concurrency.

This poster will demonstrate Pāñini's asynchronous, typed events [3]. This language feature aims to achieve a synergy between modularity in program design and concurrency for implicit-invocation design style. In this design style components are decoupled using the event abstraction. Some components, called *Subjects*, announce events. Other components, called *Observers*, express interest to receive event notifications. So subjects are able to invoke observers without becoming name-dependent on them.

SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.
ACM 978-1-4503-0240-1/10/10.

The event abstraction has been proposed before as a mean for introducing distributed concurrency, e.g. by Schmidt's reactor pattern [7], Actor-based model as in Erlang [1]. These ideas work well with a distributed memory model, where concurrent tasks are isolated. However, languages with shared memory models introduce its own challenges to concurrent execution of subjects and observers.

The novelty of Pāñini's language design, its semantics and implementation is that it addresses the challenges due to shared memory models. The resulting design thus allows programmers to modularize their programs using the implicit-invocation style and in doing so exposes safe, predictable concurrency between subjects and observers.

2. Pāñini's Language Design

Pāñini is designed as an extension of Java. Main new abstraction in Pāñini is an event type (**`event`**). For subjects, it is an abstraction of all observers and vice-versa for subjects. Thus, unlike object-oriented observer pattern [2], in Pāñini, we have a two-way decoupling of components. An example event type is shown below.

```
1 event Available {
2   Request r;
3 }
```

For a hypothetical chain-of-responsibility like scenario, this declaration gives a type to events named `Available`. This event type has a context variable of type `Request`. Context variables are reflective information about events. Just like observer design pattern, in Pāñini, components may also announce events using the **`announce`** expressions.

```
4 class Client {
5  void makeRequest (...) { ....
6   Request req = ...
7   announce Available(req);
8  }
9 }
```

Two advantages are noteworthy here. First, unlike typical usage of observer design pattern, where programmers often have to put together boiler-plate infrastructure for events, in Pāñini, event announcements are declarative. Second, Pāñini's compiler would check for correctness of event announcements and is often able to optimize announcements behind the scene to produce faster code.

Unlike observer pattern: observers must explicitly register with each subject (and thus are coupled with them), in Pāñini, registration is declarative. Following illustrates.

```
10 class Handler1 {
11   ...
12   when Available do handle;
13   void handle(Request r) {
14    if(canHandle(r)) doHandle(r);
15 }}
```

In this listing on line 12 is an example of *binding* declaration in Pāñini. This declarations says to run the method `handle` when events of type `Available` are announced.

Let us now assume a variation of chain-of-responsibility pattern where there are several handlers (`Handler1 ... HandlerN`) capable of handling request. This chain could be an implicit source of concurrency in this application, however, exploiting this concurrency in a safe manner would require analyzing these N handlers and determining if their concurrent execution can create data-races. Furthermore, to be able to understand this system's operation it would be nice to determine a sequential order of execution of these handlers and reason about the system execution that follows this order. This would avoid complications that normally arise due to interleaving between concurrent tasks.

Pāñini relieves programmers of these tasks. During compilation, Pāñini's compiler creates a summary of the effects of observers. This summary is computed by analyzing the control-flow graph to determine reads, writes, event announcements and registrations. When an observer registers at runtime, Pāñini's runtime system uses this summary to compare whether concurrent execution of this observer with other already registered observers for the same event is likely to create a data-race. This is done by comparing the effect summaries of observers. Based on this comparison a safe order of execution of observers is automatically created that maximizes available implicit concurrency in the program. Pāñini does this book-keeping during registration because registration is infrequent compared to announcement.

When an event is signalled as in the listing above, all of its observers are run in the order determined at registration. This introduces safe implicit concurrency.

2.1 Benefits of Pāñini's Design

Main benefit of Pāñini's design is that it allows developers to introduce implicit concurrency in their program (and thus achieve scalability in their applications) while producing modular software designs. Pāñini does not have locks so it is deadlock free. It uses automatic conflict detection that ensures race freedom and guarantees sequential semantics. Another key benefit is that all of the concurrency-related logic is encapsulated behind the language features. This avoids any threat of incorrect or non-deterministic concurrency, thus allowing programmers to concentrate on creating a good, maintainable modular design. Additional concurrency between modules is automatically exposed.

2.2 Advantages of Pāñini's Design over Related Ideas

Pāñini is similar to our previous work on Ptolemy [5], but Pāñini provides concurrency advantages. It is also similar to the ideas promoted for distributed concurrency [1, 7], however, Pāñini's design also accounts for conflicting effects between observers in a typical shared memory model. Compared to Jade [6] that allows implicit concurrency for explicit calls, Pāñini allows implicit concurrency for implicit invocation. Unlike message-passing languages such as Erlang [1], the communication between implicitly concurrent handlers is not limited to value types or record of value types.

3. Conclusion

In the design of Pāñini, we have developed the notion of asynchronous, typed events that are especially helpful for programs where modules are decoupled using implicit-invocation design style [4]. Event announcements provide implicit concurrency in program designs when events are signaled and consumed. We have tried out several examples, where Pāñini improves both program design and potential available concurrency. Furthermore, performance of Pāñini programs is comparable to hand-tuned explicitly concurrent programs. Thus, an important property of Pāñini's design is that, for systems utilizing implicit-invocation design style, it makes scalability a by-product of modularity.

Pāñini is available from `http://paninij.org`.

Acknowledgments. This work was supported in part by the US NSF under grant CCF-08-46059.

References

[1] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[3] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE '10: Ninth International Conference on Generative Programming and Component Engineering*, October 2010.

[4] D. Notkin, D. Garlan, W. G. Griswold, and K. J. Sullivan. Adding Implicit Invocation to Languages: Three Approaches. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, 1993.

[5] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, pages 155–179, 2008.

[6] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[7] D. C. Schmidt. Reactor: an object behavioral pattern for concurrent event demultiplexing and event handler dispatching. *Pattern languages of program design*, pages 529–545, 1995.