ICSE'15: The 37th International Conference on Software Engineering: NIER Track, Florence, Italy, May 2015.

Capsule-oriented Programming

Hridesh Rajan
Department of Computer Science
Iowa State University of Science and Technology
Ames, Iowa 50010
Email: hridesh@iastate.edu

***Abstract*—"Explicit concurrency *should be abolished from all higher-level programming languages (i.e. everything except - perhaps- plain machine code.).*" Dijkstra [1] (paraphrased). A promising class of concurrency abstractions replaces explicit concurrency mechanisms with a single linguistic mechanism that combines state and control and uses asynchronous messages for communications, e.g. active objects or actors, but that doesn't remove the hurdle of understanding non-local control transfer. What if the programming model enabled programmers to simply do what they do best, that is, to describe a system in terms of its modular structure and write sequential code to implement the operations of those modules and handles details of concurrency? In a recently sponsored NSF project we are developing such a model that we call *capsule-oriented programming* and its realization in the Panini project. This model favors modularity over explicit concurrency, encourages concurrency correctness by construction, and exploits modular structure of programs to expose implicit concurrency.**

I. Introduction

Modern software systems tend to be distributed, event-driven, and asynchronous, often requiring components to maintain multiple threads for message and event handling, and to take advantage of multicore and manycore processors to improve performance. Yet concurrent programming stubbornly remains difficult and error-prone. To address these issues, the invention and refinement of better abstractions is needed: that can hide the details of concurrency from the programmer and allow them to focus on the program logic.

The significance of better abstractions for concurrency is not lost on the research community. Some key ideas from the last two decades include: *guardians* [2], *active objects* [3], [4], and *actors* [5], all of which combine state and control within a single linguistic mechanism and use asynchronous messages for communications. These models enable implicit concurrency at the abstraction boundaries.

While the use of these actor-like entities allows concurrent execution, it does not, by default, provides data confinement and eliminate data races, since mutable objects can still be passed as messages. There have been attempts to address this issue. For example, Erlang [6] eliminates the first problem by enforcing that all data is immutable. For Scala actors, Odersky and Haller use a type system to manage ownership of objects and to ensure that references are unique [7] and Clarke *et al.* give a notion of minimal ownership for active objects [8]. We believe that two major gaps remain. First, there is an impedance mismatch between sequential and implicitly concurrent code written using actor-like entities that is hard for a sequentially trained programmer to overcome. These programmers typically rely upon the sequence of operations to reason about their programs. Second, a sweet spot between flexibility and safety has not yet been achieved, e.g. Erlang [6] provides a model where actors are fully isolated, whereas Scala provides a model with no data confinement [7]. The former model exacerbates the impedance mismatch, whereas the latter model requires additional programming mechanisms for safety, e.g. type annotations, with corresponding costs.

As part of a recent NSF project, building on our prior work on reconciling modularity and concurrency goals [9], [10], [11], we are developing a comprehensive and multifaceted approach to the challenges of concurrent programming that we call *capsule-oriented programming* [12]. A central goal of capsule-oriented programming is to provide tools to enable programmers to simply do what they do best, that is, to describe a system in terms of its modular structure and write sequential code to implement the operations of those modules using a new abstraction that we call *capsule*. A capsule is like a process in CSP [13]; it defines a set of public operations, and also serves as a memory region, or ownership domain [14], [8], for some set of ordinary objects. To the programmer, inter-capsule calls look like ordinary method calls. There are no explicit threads or synchronization locks. Capsule-oriented programs get implicit parallelism, where beneficial, due to a compilation strategy that we call *modularization-guided parallelism*. Capsule-oriented programming **eliminates** two classes of concurrency errors: race conditions due to shared data, and memory/cache consistency due to multiple cores.

In our preliminary work, we have realized basic ideas behind capsule-oriented programming in an extension of Java that we call *Panini* and have implemented a compiler for Panini by extending the industry standard OpenJDK Java compiler. It is available for download and is being actively used worldwide (See Figure 1).

We have also gained some experience with this programming model in the process of translating several already vetted benchmark suites e.g. JavaGrande (JG) [15], NAS Parallel Benchmarks [16], etc., to use capsules as the primary mechanism for concurrency. This experiment has involved several hundred thousands SLOC. It has also helped to test the robustness of our compiler and provided some informal insights into the effect of our design decisions on productivity of student programmers producing this code.

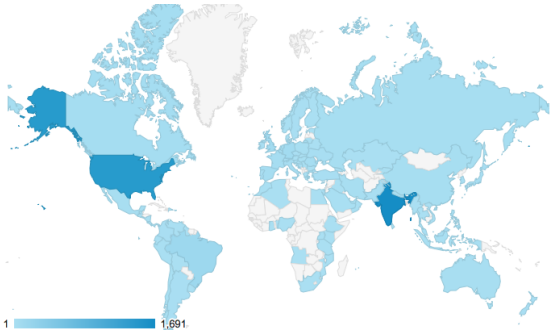

Fig. 1. The graphics above shows downloads of the Panini compiler August 2013 (public release) - Nov 2014. Panini is a capsule-oriented extension of Java.

II. Motivation

To illustrate the challenges of concurrent program design, consider a simplified navigation system. The system consists of four components: a route calculator, a maneuver generator, an interface to a GPS unit, and a UI. The UI requests a new route by invoking a `calculate` operation on the route calculator, assumed to be computationally intensive. When finished, the route is passed to the maneuver generator via method `setNewRoute`. The GPS interface continually parses the data stream from the hardware and updates the maneuver generator with the current position via method `updatePosition`. The maneuver generator checks the position against the current route and generates a new turn instruction for the UI if needed (not compute intensive).

The modular structure of the system is clear from the description above, and it is not difficult to define four Java classes with appropriate methods corresponding to this design. However, the system will not yet work. The programmer is faced with a number of nontrivial decisions: Which of these components needs to be associated with an execution thread of its own? Which operations must be executed asynchronously? Where is synchronization going to be needed? A human expert might reach the following conclusions, shown in Figure 2.

- A thread is needed to read the GPS data (lines 57-60)
- The UI, as usual, has its own event-handling thread. The calls on the UI need to pass their data to the event handling thread via the UI event queue (lines 10–14 and 17–20)
- The route calculation needs to run in a separate thread; otherwise, calls to calculateRoute will "steal" the UI event thread and cause the UI to become unresponsive (lines 33–39)
- The `ManeuverGen` class does not need a dedicated thread, however, its methods need to be synchronized, since its data is accessed by both the GPS thread and the thread doing route calculation (lines 5, 8, and 23)

None of the conclusions above, in itself, is difficult to implement in Java. Rather, in practice it is the process of visualizing the interactions between the components, in order to reach those conclusions, that is extremely challenging for programmers [17], [18].

III. Approach

The goal of capsule-oriented programming is to help sequentially trained programmers deal with the challenges of concurrent program design. Here, we present the model using the example in Figure 2.

The capsule-oriented programmer specifies a program as a collection of *capsules* and ordinary object-oriented (OO) classes. A *capsule* is an information-hiding module [19] for decomposing a system into its parts that admits implicit concurrency at its interface and a *design* is a mechanism for composing capsules together. A capsule defines a set of public operations, hides the implementation details, and could serve as a work assignment for a developer or a team of developers. Beyond these standard responsibilities, a capsule also serves as a confined memory region [20], [21], [8] for some set of standard object instances and behaves as an independent logical process [13], [5]. Inter-capsule calls look like ordinary method calls to the programmer. The OO features are standard, but there are no explicit threads or synchronization locks in capsule-oriented programming.

The example in Figure 2 contains four capsule declarations.

At first glance a capsule declaration may look similar to a class declaration, thus naturally raising the question as to why a new syntactic category is essential, and why class declarations may not be enhanced with the additional capabilities that capsules provide, namely, confinement (as in Erlang [6]) and an activity thread (as in previous work on concurrent OO languages [22], [23], [24]).

There are three main reasons for this design decision in capsule-oriented programming. First, we believe based on previous experiences that objects may be too fine-grained to think of each one as a potentially independent activity [25]. Second, we wanted to specify a program as a set of related capsules with a fixed topology, in order to make it feasible to perform static analysis of the capsule graphs; this implies that capsules should not be first-class values. Third, there is a large body of OO code that is written without any regard to confinement. Changing the semantics of classes would have made reusing this vast set of libraries difficult, if not impossible. With these design decisions, since syntactic categories are different, sequential OO code can be reused within the boundary of a capsule without needing any modification.

Compared to the explicitly concurrent system in Figure 2 (left), the capsule-oriented program in Figure 2 (right) is an implicitly concurrent program. The execution of this program begins by allocating memory for all capsule instances, and connecting them together as specified in the design declaration on lines 94-99. Recall that capsule parameters define the other capsule instances required for a capsule to function. A capsule listed in another capsule's parameter list can be sent messages from that capsule. Design declarations allow a programmer to define the connections between individual capsule instances. These connections are established before execution of any capsule instance begins.

Owing to the declarative nature of capsule-oriented features, this program is somewhat shorter compared to the program in

Java program with threads and synchronization

```
1  class ManeuverGen {
2    private Route currentRoute;
3    private Position currentPosition;
4    private UI ui;
5    public synchronized void setNewRoute(Route r) {
6      currentRoute = r;
7    }
8    public synchronized void updatePosition(Position p) {
9      currentPosition = p;
10     final Position temp = p;
11     Runnable r = new Runnable() {
12       public void run() {ui.updatePosition(temp);}
13     };
14     SwingUtilities.invokeLater(r);
15     final Instruction inst = nextManeuver();
16     if (inst != null) {
17       r = new Runnable() {
18         public void run() { ui.announceNextTurn(inst); }
19       };
20       SwingUtilities.invokeLater(r);
21     }
22   }
23   public synchronized Position getCurrentPosition() {
24     return currentPosition;
25   }
26   private Instruction nextManeuver() {/* ... */}
27 }
28 interface Calculator {void calculate(Position dst);}
29 class Shortest implements Calculator {
30   private ManeuverGen mg;
31   public Shortest(ManeuverGen mg) {this.mg = mg;}
32   public void calculate(final Position dst){
33     Thread t = new Thread(new Runnable() {
34       public void run() {
35         Route r = helper(mg.getCurrentPosition(), dst);
36         mg.setNewRoute(r);
37       }
38     });
39     t.start();
40   }
41   private Route helper(Position src, Position dst) {/* */}
42 }
43 class GPS {
44   private ManeuverGen mg;
45   public GPS(ManeuverGen mg) {this.mg = mg;}
46   public void runLoop() {
47     while (true) mg.updatePosition( readData() );
48   }
49   private native Position readData();
50 }
```

Java program, con't

```
51 class UI { /* provides updatePosition, announceNextTurn */}
52 class Navigation {
53   public static void main(String[] args) {
54     ManeuverGen mg = new ManeuverGen();
55     Calculator rc = new Shortest(mg);
56     final GPS gps = new GPS(mg);
57     Thread t = new Thread(new Runnable() {
58       public void run() { gps.runLoop(); }
59     });
60     t.start();
61     //
62     // Also create and start UI, details not shown
63     //
64   }
65 }
```

A capsule-oriented program

```
66 capsule ManeuverGen (UI ui) {     // Requires an instance of capsule UI
67   Route currentRoute = null;      // A capsule state – confined to this
         capsule
68   Position position = null;
69   void setNewRoute(Route r) { currentRoute = r; }    // A capsule procedure
70   void updatePosition(Position p) {
71     position = p;
72     ui.updatePosition(p);         // Inter–capsule procedure call
73     Instruction inst = nextManeuver();
74     if (inst != null) ui.announceNextTurn(inst);
75   }
76   Position getCurrentPosition() { return position; }
77   private Instruction nextManeuver() {/* ... */}      // A helper procedure
78 }
79 signature Calculator { void calculate(Position dst); }
80 capsule Shortest (ManeuverGen m) implements Calculator {
81   void calculate(Position dst) {
82     Route r = helper(m.getCurrentPosition(), dst);
83     m.setNewRoute(r);
84   }
85   private Route helper(Position src, Position dst) {/* ... */}
86 }
87 capsule GPS (ManeuverGen mg) {
88   void run() {
89     while (true) mg.updatePosition( readData() );
90   }
91   private native Position readData();
92 }
93 capsule UI { /* provides updatePosition, announceNextTurn */}
94 capsule Navigation {
95   design { // Specifies internal capsule instances and their interconnections
96     UI ui; ManeuverGen m; Shortest r; GPS g;        // Capsule instances
97     m(ui); r(m); g(m);              // Interconnections of capsule instances
98   }
99 }
```

Fig. 2. Programs for a simplified navigation system. Classes `Position`, `Route`, and `Instruction` are elided.

Figure 2 (left). Most importantly, this example illustrates some of the key advantages of capsule-orientation. These are:

- The programmer does not need to specify whether a given component in a system needs, or could benefit from, its own thread of execution.
- The programmer works within a familiar method-call style interface with a reasonable expectation of sequential consistency.
- All concurrency-related details are abstracted away and are fully transparent to the programmer.

IV. Ongoing Explorations

Several propeties of capsules are being investigated as part of the Panini project. We describe some of these below.

- **Modular reasoning about concurrent programs.** This is one of the most important research direction. Extant approaches do not permit modular reasoning, and we have some evidence that capsules permit modular reasoning about capsule-oriented programs [26]. This has the potential to enable **compile-time verification of concurrency-related properties** for the very first time.
- **Performant abstraction.** Performance optimization is one of the leading reasons for breaking abstraction boundaries (at least for actors). We have evidence to suggest that properties of capsules enables modular analysis to determine mapping from capsules to threads [27].
- **Capsule-oriented specification and design.** Capsule model is already more declarative compared to previous approaches. We are investigating whether further abstractions and verification at the design-level is possible.
- **Resource collection.** As a consequence of modular reasoning, capsules naturally support a simple and intuitive method of resource collection that has been a challenge [28].

- **Static analysis and unboundedness.** Capsules are carefully designed to facilitate static analysis of capsules and their interaction, e.g. for sequential inconsistency [29]. We are investigating whether certain bounds in capsule-oriented designs can be relaxed without compromising this property [30], [31].

V. Related Work

From this project's vantage point, broadly speaking, there are three kinds of approaches for concurrency: *explicit*, where programmers manually create concurrent tasks using mechanisms such as threads or fork-join pools and manage synchronization between these tasks, *implicit*, where programmers provide some hints and guidance to the compiler, often in the form of annotations or language features, and *automatic*, where the compiler is on its own to expose concurrency in a program. The capsules approach is an implicit one.

Among implicit approaches, the design of capsules was influenced by, and closely related to, the actor model [5], process calculi like CSP [13], and actor-like language features, e.g. active objects [2], [3], [4] or ambients [32]. Like abstractions in these models, capsules are also an independent ctivity. However, capsules extend these models in several ways. For example, capsules have confined semantics and thus avoid the need for integrating a separate type system with annotations [7], [8] or to make all data immutable [6]. Capsules provide in-order and transitive in-order delivery and processing semantics. Capsules provide modular reasoning [26]. None of the existing work provides all of these properties.

VI. Conclusion

A serious problem facing the current software development workforce is that software engineers have not received adequate formal training in concurrency for the last 4 decades (1970-2010). There was no apparent need to provide in-depth training in concurrency-related topics while the CPU frequency growth provided adequate scalability. A 2008 survey of 50 representative universities across the world found that only about 146 lectures ($\sim$2-3 lectures/university) were delivered on concurrency related topics [17]. Out of those, 70% were in graduate courses and the remaining were mostly in operating systems, which covered classical synchronization problems. Professional societies have noticed this gap, as is evident by a 2012 report by ACM/IEEE taskforce on computing curricula that has recently emphasized concurrency and parallelism [18]. As a result of this insufficient emphasis in curricula, we now have a workforce of which a majority is insufficiently trained in concurrent software design. While it makes great sense to develop explicit concurrency mechanisms, sequential programmers continue to find it hard to understand task interleavings and non-deterministic semantics. Thus, this research on capsule-oriented programming, if successful, will have a large positive impact on the productivity of these programmers, on the understandability and maintainability of source code that they write, and on the scalability and correctness of software systems that they produce.

References

[1] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[2] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic support for robust, distributed programs," *TOPLAS '83*, vol. 5.

[3] R. Lavender and D. Schmidt, "Active object – an object behavioral pattern for concurrent programming," in *Pattern languages of program design 2*, 1996.

[4] O. M. Nierstrasz, "Active objects in Hybrid," in *OOPSLA 1987*.

[5] G. Agha and C. Hewitt, "Concurrent programming using actors: Exploiting large-scale parallelism," in *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1985, pp. 19–41.

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

[7] P. Haller and M. Odersky, "Capabilities for uniqueness and borrowing," in *ECOOP 2010*.

[8] D. Clarke, T. Wrigstad, J. Östlund, and E. B. Johnsen, "Minimal ownership for Active Objects," in *APLAS*, 2008.

[9] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan, "Implicit invocation meets safe, implicit concurrency," in *GPCE*. ACM, 2010, pp. 63–72.

[10] H. Rajan, S. M. Kautz, and W. Rowcliffe, "Concurrency by modularity: Design patterns, a case in point," in *Onward! 2010*.

[11] H. Rajan, "Building scalable software systems in the multicore era," in *2010 FSE/SDP Workshop on the Future of Software Engineering*.

[12] H. Rajan, S. M. Kautz, E. Lin, S. L. Mooney, Y. Long, and G. Upadhyaya, "Capsule-oriented programming in the Panini language," Iowa State University, Tech. Rep. 14-08, 2014.

[13] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[14] J. Noble, J. Vitek, and J. Potter, "Flexible alias protection," in *ECOOP 1998*.

[15] L. Smith, J. Bull, and J. Obdrizalek, "A parallel Java Grande benchmark suite," in *ACM/IEEE Conf. on Supercomputing*, 2001, pp. 6–6.

[16] M. Frumkin, M. Schultz, H. Jin, and J. Yan, "Implementation of the NAS Parallel Benchmarks in Java," 2002.

[17] D. Meder, V. Pankratius, and W. F. Tichy, "Parallelism in curricula an international survey," University of Karlsruhe, Tech. Rep., 2008.

[18] ACM/IEEE-CS Joint Task Force, "Computer science curricula 2013 (CS2013)," ACM/IEEE, Tech. Rep., 2012.

[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[20] C. Grothoff, J. Palsberg, and J. Vitek, "Encapsulating objects with confined types," *ACM TOPLAS*, vol. 29, no. 6, Oct. 2007.

[21] D. G. Clarke, J. M. Potter, and J. Noble, "Ownership types for flexible alias protection," in *OOPSLA 1998*, pp. 48–64.

[22] R. Chandra, A. Gupta, and J. L. Hennessy, "COOL: An object-based language for parallel programming," *Computer '94*, vol. 27.

[23] A. Yonezawa and M. Tokoro, "Object-oriented concurrent programming," Cambridge, Mass, 1990.

[24] B. Shriver and P. Wegner, "Research directions in object-oriented programming," Cambridge, Mass, 1987.

[25] M. Papathomas, "Concurrency in object-oriented programming languages," in *Object-oriented software composition*. Prentice Hall, 1995.

[26] M. Bagherzadeh and H. Rajan, "Panini: A concurrent programming model for solving pervasive & oblivious interference," in *Modularity'15*.

[27] G. Upadhyaya and H. Rajan, "An automatic actors to threads mapping technique for jvm-based actor frameworks," in *AGERE 2014*.

[28] D. Kafura, D. Washabaugh, and J. Nelson, "Garbage collection of actors," in *ECOOP/OOPSLA*, 1990, pp. 126–134.

[29] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.

[30] Y. Hanna, S. Basu, and H. Rajan, "Behavioral automata composition for automatic topology independent verification of parameterized systems," in *ESEC/FSE 2009*.

[31] Y. Hanna, D. Samuelson, S. Basu, and H. Rajan, "Automating cut-off for multi-parameterized systems," in *ICFEM 2010*.

[32] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter, "Ambient-oriented programming in AmbientTalk," in *ECOOP'06*.