

D: A Language Framework for Distributed Programming

Cristina Videira Lopes

PhD Thesis, College of Computer Science, Northeastern University. November 1997.

© Copyright 1997 Xerox Corporation. All rights reserved.

Appendix E. User Reports

This appendix contains (I) the part of the users' final report of the space war application related to Cool and Ridl; (II) the email messages that the alpha-users have sent reporting their experience of using DJava; (III) their answers to a survey written by Prof. Gail Murphy. All documents are shown here with the permission of the people involved.

I Aspect Programming in the Space War Application

Distribution

The server and each player run on their separate machines. Remote calls occur when a player joins the game (and gets a copy of the universe from the server), a player transmits an action to the masterhelm object on the server's machine, and the player's local helm gets action an from the masterhelm. The player's clockTick method is called remotely by the server. Aside from joining the game (when information is swapped back and forth between player and server), remote communication is fairly simple.

Synchronization

Broadcasting of the ship control events (MasterHelm methods as well as Server.clockTick must be synchronized in order for all the players to see the same stream of events and keep their Universes in sync.

We achieve this by requiring that only one event can be being broadcast at a time and it must be broadcast to all the sites. The second part of this requirement is met in MasterHelm.<event> and Server.clockTick method code. The first part is guaranteed by the following synchronization conditions.

No event broadcast should occur while some player(s) has an inconsistent Universe which is the case during Server.joinGame and Server.newShip calls. We achieve this by simply stopping clock tick and message delivery processing (mutex requirement) for the duration of those calls. This is a simple but inefficient solution that is likely to change in the future.

In order to guarantee that all players are notified when a new ship has to be added to the Universe, we require selfexclusion and mutexclusion of Server.joinGame and Server.newShip.

I (Mark) find it pretty amazing (and somewhat alarming) that so many requirements are implemented by only 10-line piece of Cool spec.

Also, there is a synchronization issue that we had to solve in component code, rather than in Cool. In order to get the desired response-time we had to lower the priority of the Timer thread that calls `Server.clockTick()` method. Without that, the Java thread scheduler often kept choosing `clockTick()` thread over event threads when both were waiting on a mutex semaphore. This resulted in unacceptably long delays in event delivery.

II Users' Evaluation

Evaluation by Beth Seamans :

Anything I could say in direct response to this would repeat a bug report, a feature request, or a response to one of Gail's surveys (and usually both). John Lamping has compiled a fairly complete list of DJava shortcomings that reflect the bug report/feature request side pretty well. So rather than do that, I would like to expound briefly (if that's not an oxymoron) on a point that I think is important, and that I brought up in some recent meetings but has not really been expressed at any other time. Also, I'll be going back to the library code and re-working it for public consumption, so hopefully there will be (more) explicitly DJava-significant comments and explanations embedded. I'll let you know when that's done. If the above is insufficient, let me know and I'll give it another try.

Ridl and Cool make it much easier (faster, cleaner, more efficient) to express a concept, as long as Ridl or Cool was designed to express that concept. This has the side-effect of making it easier to think about both those concepts and the components where the concept `_used_` to be expressed. I think that our experiences coding in DJava have shown this quite well. By providing a pre-defined set of concepts or tools to the user you coerce the user into using those tools even when inappropriate (when all you have is a hammer, everything looks like a nail), but this is an inherent limitation, and all you can do is define the 'best' set of tools you can (and part of our job this summer was to help discover what the best set looks like).

So far, so good. But I would say that a large part of what makes DJava so usable and so attractive is how it simplifies the design process by exposing the concerns, and little has been done to exploit this benefit in the language(s). I have heard a lot of comments from the Guinea Pig Team to the effect of, 'if you worked in OO but had a `_really clean design_`, the benefit of AOP is not that great'. Even if that's true, the key is it's not that easy to create that 'really clean design'. We demonstrated that, too, pretty effectively. A key point (although certainly not the only point) seemed to be the need to clearly distinguish `_what_` the concerns are, and `_how well_` they are expressed by the languages provided. An optimal mapping is crucial to gaining the promised benefits. I keep bringing up the Ridl/replication muddle that Spacewar went through. More generally, design-building tools (of the kind that would be successful in OO too) would help enormously when deciding what to express in those simple, friendly languages. This could be regarded as out of the DJava scope and in the user-interface domain, but I think if DJava needs it to thrive, it should be regarded as part of the language.

I also was interested to hear Gail's comment that reading the Ridl code was very hard. The 'what' was exposed beautifully, but the motivation remained completely obscure (sometimes to the programmers, too). Without the mechanism to express that motivation,

a large part of the DJava advantage is lost. Good comments can fill that need, but good comments are as hard to find as good designs, especially since even the most diligent commenter can underestimate how much explanation is required. Building hooks into the language itself to express the 'why' as well as the 'what' could help fulfil the DJava promise.

(Beth Seamans is graduate student at Stanford University.)

Evaluation by Jared Smith-Michelson :

From my experience, the best thing about Cool and Ridl is that they greatly ease the burden of programming. Knowing nothing about the inner-workings of RMI, but having Ridl under my belt, I was able to write distributed programs. All it took was a separate Ridl file where I defined certain methods as being remote. I can only imagine the difficulties I would have had in working directly with RMI. Cool also simplified coding. Although Java has a synchronized method, it does not have explicit support for mutual exclusion. Without Cool, writing mutually exclusive methods would require the implementation of locks. The other nice feature of Cool and Ridl is that they succeed in separating concerns. Making a small change in how a field is passed or slightly reordering a mutual exclusion group is trivial in Ridl and Cool.

No distributed server-less design of the space war game was ever implemented. The main reason is that it would have been far too difficult. The synchronization issues would have been dizzying. Even with Cool and Ridl, many new abstractions would have had to have been developed to handle the communication between machines. It seems that although Ridl works very well for systems which make occasional remote calls, it does not solve the problem of making an evenly distributed, server-less, synchronizably intense network. However, this may very well be asking too much. Perhaps it's the job of a new aspect language?

One feature I'd really like to see in DJava is support for per method copy directives. In Ridl, one should be able to specify not only which fields are passed and how, but whether or not specific methods are to be called locally or remotely. Let me offer two examples which I have come across.

The first is in the spacewar game. When a new Player joins the game, a copy of the Universe is passed over. In the Universe, are copies of other Player objects. However, only a small portion of the Player object is needed, namely the id number, the name, score, etc. A large portion of the Player object need not be copied, including Console, Server, and others. Handling this problem is fairly straightforward using the current implementation of Ridl. But, it seems more natural to describe which methods will be needed, rather than which fields. For example, we wish the method `Player.getName()` to be called locally since it's called every time the Ship is painted to the screen. The `getName()` method could be declared as "passed by copy" and Ridl could then figure out which fields the `getName` method needs. The second example is regarding the distributed library. Library objects contain hashed lists of other remote Libraries to which they can forward Book Queries. Obviously, the lists need to be of refs to Libraries (you can't copy a Library!) But in or-

der to hash an Object, it needs to support the methods hashCode and equals. If the Library is regularly rehashing its indices, these methods should be remote calls. Currently, it is not possible to make certain calls remote if all you have is a graf. It would therefore be nice if Ridl supported per method copy directives.

Another feature which might be nice to have in DJava is graf to local copy conversion. Say object Foo creates an object Bar and passes it out as a graf. Then, way Foo receives the very same object Bar back again (as a graf). Could there then be some way of converting that reference to the actual Bar object, since it's in the same computation space already? Could DJava recognize that Foo already has a copy of the Bar object and that there's no need for a graf?

It would also be nice to have some more aspect languages for DJava. I was thinking along the lines of error handling, and tract debugging.

(Jared Smith-Michelson is a junior student at the Massachusetts Institute of Technology.)

Evaluation by Mark Marchukov:

I liked the way Cool handled synchronization. A short coordinator was sufficient to guarantee that all the clients in our game see the same stream of game events. On the other hand, I think that someone unfamiliar with the design of Spacewar would have a hard time understanding what purpose that coordinator really served in the program. A long comment was necessary to fully describe its purpose.

We found that we didn't need to use all the features of Cool in Spacewar, at least not in the version we finally came up with. Simple self-exclusion and mutual-exclusion declarations were enough. So I think that Spacewar was not really coordination-intensive.

Ridl with its convenient support for global references was useful at the initialization stages of the game when the references between objects have to be set up. It also pushed us towards a design that looked like there was no distribution: we used global references to deliver messages to individual ships across the network as opposed to delivering them to *clients* that would in turn deliver them to local ships. But since RMI is grossly inefficient, our program was slow too and no part of Ridl could help us make it faster by sending less data over the network. That's a pity.

And lastly, the design of Spacewar that we used in the distributed version was heavily based on replication of objects and keeping replicas on different sites in sync. Ridl offered us no direct control over replication and distributed synchronization aspects. After all, it wasn't its job, it wasn't designed for that. This is I think why the DJava version of Spacewar has approximately the same complexity as the Java-only version that we wrote later.

(Mark Marchukov is a graduate student at the University of Virginia)

III Prof. Murphy's survey

The part of the survey that is shown here is the following:

A Survey for the AOP Trailblazers
July 23, 1997

This survey consists of eleven main questions (some with sub-questions). Your answers need not be long, but try to refer to concrete examples (through pointers to code, etc.) whenever possible.

Using Aspect-Oriented Programming.

1. Please estimate the amount of time you spend thinking about aspects when:
 - i. designing a component
 - ii. implementing a component
 - iii. Are there any particular components/aspects for which the time has been much different than the time stated above? If so, which ones?
2. Are the components you have written independent from their aspects (e.g., could you remove the aspects and still have functioning components, is there partial dependence, etc.). If applicable, describe an instance in which there was coupling between the component and the aspect. (i.e., the component was written differently because of the aspect)
3. In your first report, you described four different "processes" for AOP design. Which one most reflects the way you are doing Aspect-Oriented design? (*NOTE: not sufficient context for understanding this question; the answers were omitted*)
4. Do you find it (easy/moderately difficult/difficult/impossible) to read someone else's aspect code and understand it? How do you resolve any problems you experience in understanding an aspect?
5. How do you decide if your aspect code is working? (e.g., do you run test cases, reason through the code, etc.?)
6. What procedures do you use to debug your DJava code?

Answers by Beth Seamans:

1.
 - i. designing a component
some
 - ii. implementing a component
very little

'Some' is hard to define. Most of the time, the aspects did not require an inordinate amount of attention. When Ridl reality did not match our understanding, or when a Ridl operation supporting a clean design did not exist, we were forced to spend much more time. Regarding the implementation, we usually did a detailed design of the class interface structure, where most of the aspect issues crop up, so the implementation was fairly routine.

We have been thinking that there may be more sophisticated and efficient ways to use Cool, and that will take more consideration.

- iii. As noted above, the unsolved Ridl bugs and unimplemented Ridl feature requests require more time on the design.
- 2. I think the components are independent, but would be quite inefficient since there is currently a lot of duplication of information across machines.
The Masterhelm/Helm relationship is explicit in the component code, and arises from the distributed aspect.
- 4. The aspect code, since it is so minimal, is usually easy to understand. It can be more difficult to see the 'why', but the 'what' is clear. [Resolving problems of understanding] Talking and drawing diagrams. Repeat as necessary. Having four people together, with ready access to expert help, makes the confusion/clarification cycle very small.
- 5. We reason through the code and play the game to test. Since it is non-deterministic, it's difficult to make sure all bases are covered, but quite a bit of hands-on playing gets done.
- 6. We try to run it without the aspects first, if at all possible. The debugging process is mostly a matter of isolating through printed debug statements.

Answers by Jared Smith-Mickelson :

1

- i. For the most part, I don't think too much about aspects while designing a component. Although, that's not to say I ignore them altogether. Sometimes, knowing that I have Cool and Ridl as tools effects the design of the component. For example, in the spacewar game, there is a Console object which paints graphics to the screen and a Universe object which maintains the positions and velocities of the SpaceObjects. The Server periodically sends clock ticks to the Universe so that the game stays synchronized. The problem we were facing was how to notify the Console when the Universe was ticked. We were thinking along the lines of having the Universe tick the Console or having the Server tick both, when it occurred to us that the best solution was to write some Cool code. We realized that Console doesn't even need a clockTick method. All it needs to do is continuously redraw the screen. A Cool file could describe a guard which would suspend the Console until the Universe was updated. This would keep synchronization concerns where they belong, in a Cool file.
- ii. During implementation, aspects play a stronger role. This is mainly due to the fact that most of the gritty details are worked out here. When implementing components, I find myself thinking about remote argument passing quite often. I'm concerned about which calls should be remote, and whether or not object foo needs a reference to bar or a copy of bar. For the most part, I find myself thinking about the distribution aspect of the program. I usually save synchronization for the end, for I have found that most of the time method exclusion can be added without having to change underlying design or implementation.
- 2. I wouldn't call them independent. They'd still compile, but their functionality would likely be corrupted. However, I see nothing wrong with this. After all, components are dependent on other components, aren't they? Therefore, why can't they depend on aspects? To me, aspects and components are means by which to modularize the code, not decouple it. For example, in the Console/Universe example above, the program wouldn't work properly without the Cool

code. The Console, without sleeping or suspending its own thread, would hog the processor and virtually freeze everything else.

In version 0.3 of the spacewar program, there is a very complex protocol for the joining of Players to the game. This protocol lends itself almost entirely to what Ridl could and could not do at the time of implementation. For example, in a couple places, it was necessary for the Server to create a global reference to a local object. Since Ridl could not do this, objects had to be passed back and forth between remote spaces using methods not present in the original design.

4. I find aspect code to be extremely clear and easy to read. I can only imagine the nightmare of reading through woven Java output looking for deeper meaning behind the slew of Locks and TraversalPatterns.
5. When it comes to deciding whether my code is working properly, I treat aspects the same as I do components. First I run the program, looking for correct behavior. If something is not working the way I anticipated, I reason through the code. If I can't determine what's going wrong, I write smaller test cases, in an attempt to get to the root of the problem.
6. Usually, the compiler or runtime error messages will suffice. But if I have to, I put in print-outs to trace the execution path. It would be nice to have a debugger to step through the code, but alas...

Answers by Mark Marchukov:

1.
 - i. It depends on what you mean by "aspect". If distribution is an "aspect", then 60%. This includes discussing both .Cool and .Ridl declarations and component structure and actions. If only .Cool and .Ridl declarations are "aspects", then 20%.
 - ii. 20%. We are usually doing a very detailed design. I didn't include the time we spend to find work-arounds for weaver bugs.
2. Looks like your definition of "aspect" is close to the second one above... Let's see... Server and MasterHelm have to be synchronized. They will function without their synchronization aspects, but not in this program. Their Ridl parts can be removed. Not so for Helm that relies on the fact that setShip(s) gets a copy of s, not a reference. Universe could function without its coordinator, in a single-threaded program. Player would be unusable without its Ridl part. Player is a good example. Its remoteClone() method simply returns -this-. And it is all up to its Ridl aspect to determine what and how is returned. It is meaningless to have a method that returns -this- in a class, unless it has a distribution aspect.
4. Aspect code that we write is very small. I wouldn't even call it "code" because it doesn't have a flow of control in it. These are declarations. They are easy to understand. However, to understand what the program will do when the component code corresponding to this aspect specification executes, one must understand the *component* code. And this is not always easy. This would require looking at both component code and aspect specs and going "a-ha! when I make this call the result comes by copy, these fields are returned by global reference and these are skipped. That's why I'm getting a null pointer exception!"
5. I don't think I have had to look at an unfamiliar piece of aspect code so far. But if I had, I would have done what I described above: looked at both the aspect and its component(s).
6. System.out.println()

