Problems with Concurrent Dock Use in Admiral

Amir Kamil
Computer Science Division, University of California, Berkeley
kamil@cs.berkeley.edu

UCB-AK05

December 17, 2007

1 Introduction

In the Admiral language, it is currently possible to concurrently use the same dock in different move statements. This can be the case if two statements in a parallel block use the same dock, as in Figure 1, or in the future, if dock sharing is allowed between different named blocks. In this document, I discuss the implications of concurrent dock usage.

2 Implementation of Sequencing

The Admiral compiler will enforce sequencing between two move statements by ensuring that there is some form of dependency between them that prevents the second from executing until the first has completed. In many cases, this will be accomplished using tokens. For the first sequential block in Figure 1, it is sufficient to program dock `b` to send a token to dock `c` when data has arrived at `b` and to program `c` to wait for a token before sending data to `d`, as shown in the left of Figure 2. The assembly code to implement the second sequential block is similar and is shown in the right of Figure 2.

3 Effect of Concurrent Dock Use on Sequencing

The sequencing strategy used by the Admiral compiler can be broken by concurrent use of a dock. For the example in Figure 1, the compiler can order the two instructions at dock `c` arbitrarily, since the program does not specify a sequence. Figure 3 shows the complete assembly code resulting from one such ordering. However, the instructions at dock `c` have no way of determining where tokens arrive from. Thus, if a token arrives from dock `f`, signaling the completion of the Admiral statement `e -> g`, before the statement `a -> b` completes, the first instruction at dock `c` will be triggered. This starts the execution of the statement `c -> d`, violating the sequencing requirement that `a -> b` finish execution before `c -> d` starts.

```
{
  {
    a -> b ;;
    c -> d
  } ||
  {
    e -> f ;;
    c -> g ;;
  }
}
```

Figure 1: Admiral code that concurrently uses dock `c`.

```
a:                                e:
  take, sendto b;                   take, sendto f;
b:                                f:
  take, deliver, ack c;             take, deliver, ack c;
c:                                c:
  wait, take, sendto d;             wait, take, sendto g;
d:                                g:
  take, deliver;                    take, deliver;
```

Figure 2: Implementation of the two sequential blocks in Figure 1 in Fleet assembly.

```
a:
  take, sendto b;
b:
  take, deliver, ack c;
c:
  wait, take, sendto d;
  wait, take, sendto g;
d:
  take, deliver;
e:
  take, sendto f;
f:
  take, deliver, ack c;
g:
  take, deliver;
```

Figure 3: A possible implementation of the code in Figure 1 in Fleet assembly.

Of course, for the example in Figure 1, it can be pointed out that the program already contains a race condition, since it does not specify whether the first item from `c` should go to `d` or `g`. This, however, is obvious from the source code and may not be a problem in reality, whereas the sequencing violation is not obvious from the code and is likely to affect correctness of the program.

4 Solutions

Since concurrent dock use violates sequencing in a subtle and non-obvious manner, it is unreasonable to expect the Admiral programmer to solve the problem. Instead, the solution must be in the language or compiler itself.

The simplest solution is to forbid the concurrent use of any dock. This can be done by making it illegal to mention a dock in different statements within a parallel block, and by introducing restrictions on docks exported by virtual ships or shared between named blocks to prevent the docks from being used concurrently. This solution, while simple, is also very restrictive, as it forbids concurrent use of a dock even if it does not violate sequencing. A more relaxed solution, such as specifying that "the results are undefined" if a dock is used concurrently, may be more appropriate. This can be coupled with a compiler warning when a sequencing violation is possible.

A compiler-level solution is also possible, though more complex. Instead of dispatching conflicting instructions to a concurrently-used dock, the instructions can be stored somewhere else, such as in separate FIFOs, and the sequencing token can be used to dispatch the instruction instead of triggering it. For the example in Figure 1, the two instructions at dock `c` can be stored in FIFOs $F1$ and $F2$. Then the token from dock `b` can be used to trigger a move from the output of $F1$ to the instruction destination of `c`, and the token from dock `f` to trigger a move from the output of $F2$ to the instruction destination of `c`, as shown in Figure 4. This solution requires the compiler to determine all possible concurrent uses of a dock, so if a dock can be shared between two named blocks or between an instance of a virtual ship and its user, something similar to interprocedural analysis would be required. Thus, it may be useful to prevent dock sharing in this solution as well.

```
{c: take, sendto d}:
  sendto F1.in;
{c: take, sendto g}:
  sendto F2.in;
F1.in:
  take, deliver;
F2.in:
  take, deliver;
F1.out:
  wait, take, sendto c$ins;
F2.out:
  wait, take, sendto c$ins;
a:
  take, sendto b;
b:
  take, deliver, ack F1.out;
d:
  take, deliver;
e:
  take, sendto f;
f:
  take, deliver, ack F2.out;
g:
  take, deliver;
```

Figure 4: A correctly-sequenced implementation of the code in Figure 1, using two FIFOs. Here, the syntax "{...}" denotes an instruction literal, and "`c$ins`" denotes the instruction destination of dock `c`.