Implementing π-calculus style actions

D.R.W. Holton

April 2008

1 Introduction

This technical report describes one technique for implementing π-calculus style actions in a programming language. It first attempts to clarify the nature of actions, then gives requirements a primitive in a programming language must satisfy if it is to be used as the basis for the implementation of actions. Finally an example is given of how actions may be implemented in Ada.

2 Understanding Actions

One of the most mysterious aspects of the π-calculus is exactly **what** an action represents. We understand that when an action and its complement are offered, a synchronous reaction occurs between two agents which then evolve to new states. But what exactly are these actions?

The key to understanding actions is to question the **ownership** of an action. In the context of this discussion, ownership is determined by who controls the use of an action. To clarify, an action may be defined locally within an agent, which defines its scope and lifetime, however that agent does not control how the action is used, nor which of the agents that has the action in scope may use it. The example below perhaps clarifies this:

$$Server \quad = \quad \text{new } put, get \quad Gatekeeper \,|\, Buffer \,|\, Processor$$

The bound actions put and get are restricted to the agents $Gatekeeper$, $Buffer$ and $Processor$ but there are no controls about which, if any, of these agents may use them. Also, since actions may be passed as parameters it is possible that agents other than those named may also use these actions.

So who owns an action? The simple answer is: itself. No agent owns an action, it is shared among potentially many agents. For once, this is in accord with everyday notions of actions such as playing chess: two players (agents) may play a game of chess, but neither owns "chess". To take the analogy further, internet chess servers[1] allow players who do not know each other to perform the "chess" action; alternatively two players may share a private game.

Now that ownership has been understood, the nature of actions is clearer: actions are atomic events that cause agents to synchronise then evolve – and of course this is obvious when re-reading the primary source [2] where the starting point of the development is classical automata theory.

3 Choosing the Correct Programming Language Primitive

Given the understanding of actions gained above, it is now possible to list the requirements for a programming language primitive for implementing actions:

[1] E.g. `www.freechess.org`

- it should encapsulate instructions to implement the action;
- those instructions must be executed atomically;
- it must permit synchronisation of agents;
- any two agents may synchronise via any action that is in scope;
- it must be possible to restrict the scope of actions, yet also allow scope extrusion;
- it must be possible for actions to be created and destroyed dynamically.

Actions are commonly described as a channels (e.g. [5, page 5]) and this has influenced how they are implemented (e.g. [3, 6]). However an action is more flexible than channels provided by many programming languages (e.g. [1]), particularly with respect to scope and lifetime, as well as allowing synchronisation between agents without either knowing the identity of the other.

A more flexible approach to implementing actions is to use subprograms (such as method calls in an object) as described in [4], however the actions in this approach are burdened with extra responsibilities, such as detecting when they are being used inappropriately, which are not their concern. Furthermore, encapsulating those subprograms in a language primitive that can have instances created and destroyed as required and permits references to instances to be passed as parameters allows the requirements above to be satisfied – provided the language supports concurrency control primitives.

A class in an object oriented programming language would be satisfactory, with the ubiquitous semaphore used to provide synchronisation and atomicity. In the same way that the π-calculus distinguishes between actions and their complements, the class would provide methods denoted `Put` and `Get`, allowing values to be transferred between agents[2]. Semaphores would be used to enforce synchronisation: an agent cannot take a value before it has been provided; an agent should wait until the value it has provided has been taken. This "lowest common denominator" approach ensures that actions may be implemented in any language that provides classes and has access to semaphores – modest requirements that are satisfied by many languages. However, as seen below, if a language has richer features then these may be used instead. For example, actions may be implemented in Java using guarded blocks within synchronised methods.

4 Implementation in Ada

Ada provides language primitives that greatly ease the construction of concurrent programs. Of particular interest is the protected object, a language primitive related to the monitor which guarantees mutual exclusion when the data it encapsulates is updated. It also provides condition synchronisation by the use of barriers, and a requeuing mechanism to allow a task to release mutually exclusive access to a protected object while it waits for a condition to become true. Protected objects have an "internal progress first" rule, meaning that tasks which have been suspended while executing within an object will be resumed before a new task is allowed to enter that object. In addition Ada permits (encourages) subprogram definitions to be overloaded, allowing subprograms with different parameter profiles but using the same name to be written – this allows an action to be used both for synchronisation (no parameters) and for data exchange. While Ada does not allow access types (pointers) as parameters to entries, with some sleight of hand it is possible to pass references to actions as parameters.

The implementation proceeds as follows:

[2]This scheme allows data to be both output and input by each agent, and also allows a value to be agreed between the agents – useful in applications such as cryptographic key agreement.

1. Define a protected type with entries Put and Get overloaded for each parameter profile used in the specification and local variables to store the parameters for each action;

2. For each Put entry, define a private entry Wait_Put_i to allow the calling task to be suspended until the value it has deposited has been taken;

3. Each triple ⟨Put, Get, Wait_Put_i⟩ of overloaded entries has an associated boolean variable Enter_i used in their entry barriers to ensure that the calling task will proceed only when the appropriate condition has been met;

4. A task calling a Put entry will only proceed when there is no previously deposited value to be taken, and after a copy of the value has been stored the calling task opens the barrier to allow that value to be taken and is requeued until it is taken.

5. A task calling a Get entry will only proceed when the value it requires has been deposited – i.e. the barrier is initially closed – and before completing the entry it will open the barrier to allow the task which deposited the value it has taken to proceed;

It is convenient to combine all possible Put and Get entries in a single protected type so that there is a single type Action, instances of which can be created as required – the fact that some instances contain entries that are never called is of no concern.

It may appear odd that it is possible to use a single boolean variable for three barriers since two barriers must be open at once. However, due to the internal progress first rule, it is correct to allow the barriers for Put and Wait_Put_i to be open at the same time since a task waiting at Wait_Put_i will proceed before the first task waiting at Put is allowed access to the protected object.

The sleight of hand mentioned above requires that references to protected objects are translated to raw memory addresses when they are passed as parameters, then translated back to a reference before being used. Ada's generic System.Address_To_Access_Conversions package is used for this purpose and subprograms to perform the translation are provided. While this is not an ideal solution, it does at least have the virtue that the details of the translation procedure are hidden.

This procedure is best illustrated with an example. Isn't it great

$$
\begin{aligned}
Client &\stackrel{def}{=} \text{new reply } \overline{port_1}\langle 12\rangle.port_1.Client \\
&\quad + \ \overline{port_2}\langle reply\rangle.reply.Client \\
Server_1 &\stackrel{def}{=} port_1(q).\overline{port_1}.Server_1 \\
Server_2 &\stackrel{def}{=} port_2(reply).\overline{reply}.Server_2 \\
Machine &\stackrel{def}{=} \text{new } port_1, port_2 \ Client \mid Client \mid Server_1 \mid Server_2
\end{aligned}
$$

The specification for the protected type follows – note the different versions of the Put and Get entries and the associated private entries and boolean variables. The numerical suffix on the private declarations indicates associations between variables and entries – the first pair of Put and Get entries is associated with the suffix 0 and so on.

```
with System.Address_To_Access_Conversions;

use System;

package Actions is

   protected type Name is
      entry Put;
      entry Get;
```

```
      entry Put(q : Natural);
      entry Get(q : out Natural);
      entry Put(reply : Address);
      entry Get(reply : out Address);
   private
      My_q : Natural;
      My_reply : Address;
      entry Wait_Put_0;
      entry Wait_Put_1;
      entry Wait_Put_2;
      Enter_0, Enter_1, Enter_2 : Boolean := True;
   end Name;

   package Address_To_Name is
      new System.Address_To_Access_Conversions(Name);

   subtype Action is Address_To_Name.Object_Pointer;

   function Translate(Nick : Address) return Action;
   function Translate(Nick : Action) return Address;

end Actions;
```

The protected type is implemented as follows. Notice how the boolean variables are modified by each operation to allow the other partner in the synchronisation to proceed.

```
package body Actions is

      protected body Name is

      entry Wait_Put_0 when Enter_0 is
      begin
         null;
      end Wait_Put_0;

      entry Put when Enter_0 is
      begin
        Enter_0 := False;
         requeue Wait_Put_0;
      end Put;

      entry Get when not Enter_0 is
      begin
         Enter_0 := True;
      end Get;

      entry Wait_Put_1 when Enter_1 is
      begin
         null;
      end Wait_Put_1;

      entry Put(q : Natural) when Enter_1 is
      begin
         Enter_1 := False;
         My_q := q;
         requeue Wait_Put_1;
      end Put;
```

```
      entry Get(q : out Natural) when not Enter_1 is
      begin
         Enter_1 := True;
         q := My_q;
      end Get;

      entry Wait_Put_2 when Enter_2 is
      begin
         null;
      end Wait_Put_2;

      entry Put(reply : Address) when Enter_2 is
      begin
         My_reply := reply;
         Enter_2 := False;
         requeue Wait_Put_2;
      end Put;

      entry Get(reply : out Address) when not Enter_2 is
      begin
         reply := My_reply;
         Enter_2 := True;
      end Get;
   end Name;

   function Translate(Nick : Address) return Action is
   begin
      return Address_To_Name.To_Pointer(Nick);
   end Translate;

   function Translate(Nick : Action) return Address is
   begin
      return Address_To_Name.To_Address(Nick);
   end Translate;

end Actions;
```

Finally an implementation of the *Machine* above is given, showing the declaration of the actions and the clumsy translations required when passing actions as parameters. The use of a local block in the Client task is to illustrate the dynamic creation and destruction of actions. Also note that the non-deterministic choice in the Client agent has been replaced by a deterministic choice in the Client task (and the unbounded looping has been replaced by a finite number of iterations in all tasks).

```
with System, Actions;
use System, Actions;

procedure Machine is

   port1, port2 : Action := new Actions.Name;

   task type Client;
   task Server1;
   task Server2;
   Client1, Client2 : Client;
```

```
   task body Client is
   begin
      for I in 1..10 loop
         declare
            reply : aliased Action := new Actions.Name;
         begin
            if I mod 2 = 0 then
               port1.Put(12);
               port1.Get;
            else
               port2.Put(Translate(reply));
               reply.Get;
            end if;
         end;
      end loop;
   end Client;

   task body Server1 is
      Q : Natural;
   begin
      for I in 1..10 loop
         port1.Get(Q);
         port1.Put;
      end loop;
   end Server1;

   task body Server2 is
      reply : aliased Action;
      Alias : Address;
   begin
      for I in 1..10 loop
         port2.Get(Alias);
         reply := Translate(Alias);
         reply.Put;
      end loop;
   end Server2;

begin
   null;
end Machine;
```

5 Analysis and Conclusions

As is to be expected, much of the non-determinism of the π-calculus specification has been eliminated in the implementation. The obvious replacement of non-deterministic choice is one example. Another example is when several agents are trying to perform the same action, such as $\mathtt{port}_1$ in the example above: the choice of which agents react is non-deterministic, meaning that any agent may suffer starvation. In the implementation there is a strict FIFO ordering of requests and starvation does not occur.

As it stands, this work does not address the significant complexities that arise from the non-deterministic choice operator. The issue was evaded in the implementation of the `Client` by implementing a deterministic choice, but this is not always possible or desirable. For example a

common idiom is for a server to offer a range of actions, each associated with a different service, and to serve clients on demand. The usual implementation of this in Ada is for the server to own a number of entries and offer them in a selective accept statement. This solutions goes against the principle that actions are not owned by agents, so is not applicable. Although more work is required, it is likely that such situations will require a busy-waiting solution, which is inelegant[3].

References

[1] A. Burns. *Programming in Occam 2*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1987.

[2] R. Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, 1999.

[3] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, chapter 15. MIT Press, 2000.

[4] N. Rodrigues and L. S. Barbosa. Prototyping behavioural specifications in the. net framework. *Proc. 7th Brazilian Symposium on Formal Methods*, pages 108–118, 2004.

[5] P. Sewell. *Applied π – A Brief Tutorial*. University of Cambridge Computer Laboratory, 2000.

[6] Lucian Wischik. New directions in implementing the pi calculus. In *CaberNet Radicals Workshop*, 2002.

[3] Not to mention inefficient!!