Communicating Process Architectures 2008
Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.)
IOS Press, 2008

Two-Way Protocols for occam-π

Adam T. SAMPSON [1]

Computing Laboratory, University of Kent

Abstract. In the occam-π programming language, the client-server communication pattern is generally implemented using a pair of unidirectional channels. While each channel's protocol can be specified individually, no mechanism is yet provided to indicate the relationship between the two protocols; it is therefore not possible to statically check the safety of client-server communications. This paper proposes *two-way protocols* for individual channels, which would both define the structure of messages and allow the patterns of communication between processes to be specified. We show how conformance to two-way protocols can be statically checked by the occam-π compiler using Honda's session types. These mechanisms would considerably simplify the implementation of complex, dynamic client-server systems.

Keywords. Client-server, Concurrency, occam-π, Protocols, Session types

Introduction

The occam-π process-oriented programming language supports very large numbers of lightweight processes, communicating using channels and synchronising upon barriers. It has roots in CSP and the π-calculus, making it possible to reason formally about the behaviour of programs at all levels from individual processes up to complete systems.

In a process-oriented system, it is very common to have client-server relationships between processes: a server process answers requests from one or more clients, and may itself act as a client to other servers while processing those requests (see figure 1). The *client-server design rules* [1] allow the construction of client-server systems of processes that are guaranteed to be free from deadlock and livelock problems.

The client-server pattern has proved extremely useful when building complex process-oriented systems. Server processes fill approximately the same role as objects in object-oriented languages; indeed, OO languages such as Smalltalk use message-passing terminology to describe method calls between objects. However, process-oriented servers avoid many of the concurrency problems endemic in OO languages, and their interfaces are more powerful: a client-server communication may be a *conversation* containing several messages in both directions, not just a single request-response pair.

Most non-trivial occam-π programs today make some use of the client-server pattern, with communication implemented using channels. However, while occam-π allows the protocol carried over an individual channel to be specified and checked by the compiler, it does not yet provide any facilities for checking the protocols used across two-way communication links such as client-server connections.

In this paper, we will first describe occam-π's existing protocol specification facilities, and how they are currently used to implement client-server communications. We will then examine other implementations of two-way communication protocols, and how they can be

[1]Corresponding Author: *Adam Sampson, Computing Laboratory, University of Kent, CT2 7NF, UK*. Tel.: +44 1227 827841; E-mail: `A.T.Sampson@kent.ac.uk`.

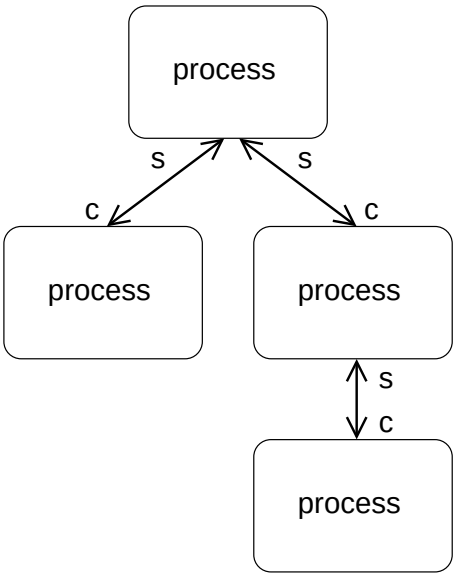

Figure 1. Client-server relationships between processes

formally specified using session types. Finally, we will describe how two-way protocols could be specified and implemented in occam-π, and discuss some possible syntax for them.

1. Unidirectional Protocols

occam-π's channels are unidirectional and unbuffered, and the order of messages permitted over each channel is specified using a protocol. The compiler checks that processes using channels adhere to their protocols. Protocols have been present in the language since occam 2 [2].

A *simple protocol* is just a list of types which will be sent in sequence over the channel.

```
PROTOCOL REPORT IS INT; REAL32:
CHAN REPORT c:
c ! 42; 3.141
```

A *variant protocol* allows choice between several simple protocols, each identified by a unique tag; each communication over the channel is preceded by a tag that indicates the simple protocol to be followed.

```
PROTOCOL COLOUR
  CASE
    rgb; REAL32; REAL32; REAL32
    palettised; INT
:
CHAN COLOUR c:
SEQ
  c ! rgb; 0.3; 0.3; 0.0
  c ! palettised; 42
```

occam-π introduced the idea of *protocol inheritance* [3], which allows the tags from one or more existing variant protocols to be incorporated into a new variant protocol. A reading end of a channel carrying one of the included protocols may be used as if it were a reading end of a channel carrying the new protocol; for writing ends, the opposite applies.

```
PROTOCOL PRINT.COLOUR EXTENDS COLOUR
  CASE
    cmyk; REAL32; REAL32; REAL32; REAL32
:
CHAN PRINT.COLOUR c:
SEQ
  c ! rgb; 0.3; 0.3; 0.0
  c ! cmyk; 0.1; 0.4; 0.1; 0.0
```

Several channels may be grouped into a *channel bundle* [4]. The ends of a channel bundle are *mobile*: they may be sent around between processes, allowing the process network to be dynamically reconfigured at runtime. The channels inside a bundle may be used as if they were regular channels.

```
CHAN TYPE DISPLAY
  MOBILE RECORD
    CHAN COORDS in?:
    CHAN COLOUR out!:
:
DISPLAY! end:
INT c:
SEQ
  end[in] ! 2.4; 6.8
  end[out] ? CASE palettised; c
```

2. Client-Server Communication in occam-π

Client-server communications are currently implemented in occam-π using a pair of channels: one carries *requests* from the client to the server, and the other carries *responses* from the server to the client. The two channels are usually packaged inside a channel bundle. We can use this approach to specify a client-server interface to a random-number generator, which will attempt to roll an N-sided die for you, and either succeed or drop it on the floor:

```
PROTOCOL DIE.REQ
  CASE
    roll; INT
    quit
:
PROTOCOL DIE.RESP
  CASE
    rolled; INT
    dropped
:
CHAN TYPE DIE
  MOBILE RECORD
    CHAN DIE.REQ req?:
    CHAN DIE.RESP resp!:
:
```

The `req` and `resp` channels carry requests and responses respectively, each with their own protocol. In this case, a `roll` message from the client would provoke a `rolled` or `dropped` response from the server; a `quit` message would cause the server to exit with no response. To use this process, a client need only send the appropriate messages over the channels in the bundle:

```
PROC roll.die (DIE! die)
  SEQ
    ...  obtain die

    die[req] ! roll; 6
    die[resp] ? CASE
      INT n:
      rolled; n
        ...  rolled an 'n'
      dropped
        ...  dropped the die

    die[req] ! quit
:
```

The syntax for defining and using client-server interfaces is rather clumsy. Each client-server interface requires two protocols and a channel bundle type to be declared. The protocol names – **DIE.REQ** and **DIE.RESP** in this case – are usually only used within the channel bundle definition. When sending messages over a client-server interface, the name of the channel being used must always be specified, even though it is unambiguous from the direction of communication whether the **req** or **resp** channel should be used.

More seriously, occam-π provides no facility for specifying the relationship between the two protocols in a client-server interface. By convention, the programmer writes a comment saying "replies rolled or dropped" next to the definition of **roll**, but this is only useful to humans. The compiler cannot check that the processes using the channel bundle are correctly ordering messages between channels. For example, a process like this correctly follows the protocol on each individual channel:

```
SEQ
  die[req] ! roll; 6
  die[req] ! roll; 6
```

However, it would deadlock because the server expects to only receive a single **roll** message before sending a response. At the moment, it will be accepted by the compiler without complaint.

The vast majority of channel bundle definitions in existing occam-π code are client-server interfaces like **DIE**. Providing a more convenient language binding for client-server interfaces would not only simplify many programs, but also allow the compiler to detect more programmer errors at compile time.

3. Related Work

Facilities for specifying two-way communication are present in some other process-oriented languages.

The draft occam 3 language specification [5] described a *call channels* mechanism built on top of channel bundles; this provided a way of declaring channel bundles that were used for call-response communications. The declaration of a call channel therefore implicitly defined protocols to carry the parameters and results of a procedure. A call channel named **cosine** with a single input parameter and a single result would be defined as:

```
CALL cosine (RESULT REAL32 result, VAL REAL32 x):
```

The suggested syntax made clients look like procedure calls, and servers look like procedure declarations:

```
PAR
  cosine (cos.pi, 3.141)
  ACCEPT cosine (RESULT REAL32 result, VAL REAL32 x)
    result := COS (x)
```

Since **ACCEPT** is implemented as a channel input for the parameters, followed by a channel output for the results after the block is complete, it is possible to use it as a guard in an **ALT**. The same idea has been implemented in other process-oriented frameworks such as JCSP [6].

Call channels are a useful abstraction for programmers transitioning from the object-oriented world, since they make calls to a server look like method calls upon an object. However, they only allow a single request and response; they do not provide the richer conversations afforded by protocols.

The Honeysuckle language provides facilities for easily composing client-server systems, with interfaces being defined as *services* [7]. Of particular interest here are *compound services*, which allow a server's behaviour to be specified using a subset of Honeysuckle including communication, choice and repetition constructs:

```
service class Console :
{
  ...

  sequence
    receive command
    if command
      write
        acquire String
      read
        sequence
          receive Cardinal
          transfer String
}
```

This notation is very powerful; it allows arbitrary conversations between a client and server to be precisely specified. It is, however, possible to specify a protocol that cannot be statically verified by using repetition with a count obtained from a channel communication. Such protocols may require runtime checks to be inserted by the compiler if the repetition counts cannot be statically determined.

Honeysuckle's services provide a convenient, flexible way of specifying client-server interfaces; we would like to provide a similar facility in occam-π.

4. Session Types

Session types [8] provide a formal approach to the problem of specifying the interactions between multiple processes, by allowing communication protocols to be specified as types. The type of a communication channel therefore describes the sequence of messages that may be sent across it. For example, a channel with the session type

$$foo! \,.\, bar?$$

can be used to send ("!") the message foo, then receive ("?") the message bar; the "." operator sequences communications.

Session types can also specify choice between several labelled variants using the "|" operator. For example,

$$(left! \ . \ INT!) \mid (right! \ . \ BYTE!)$$

can be used to either send $left$ followed by an integer, or $right$ followed by a byte.

When checking the correctness of a process, a session-type-aware compiler will update the type of each channel as communications are performed using it. For example, if a channel's session type is initially $foo! \ . \ bar? \ . \ baz?$, after it is used to send the message foo, its session type will be updated to $bar? \ . \ baz?$.

Session types were originally defined in terms of the π-calculus, but can also be applied to network protocols, operations in distributed systems, and – most interestingly for our purposes – communications between threads in concurrent programming languages.

Neubauer and Thiemann [9] describe an encoding of session types in Haskell's type system, representing communication operations using a continuation-passing approach. Session types may be defined recursively, which is convenient for specifying protocols containing repetition or state progression – for example, a type may be defined as several operations followed by itself again. The specifications are applied to sequences of IO operations, such as communications on a network socket; there is no discussion of their application to local communication, although the same approach could be used to sequence communication between threads.

The L_{doos} language [10] integrates object-oriented programming and session types. Its session type specifications cannot contain branching or selection, but they support arbitrary sequences of communications in both directions, making them more flexible than simple method calls.

Vasconcelos, Ravara and Gay [11] give operational semantics and type-checking rules for a simple functional language with lightweight processes and π-calculus-style channels, where channel protocols are specified using session types. Its session types may be defined recursively, and may include choice between several labelled options. It notes that aliasing of channels can introduce consistency problems, since operations may affect one alias and not update the session type of the others. It demonstrates that session types can be applied effectively to communication between local concurrent processes.

The SJ language [12] extends Java with *session sockets* that are conceptually similar to TCP sockets (and are implemented using TCP), but over which communication takes place according to protocols which are defined using session types. It supports conditional and iteration constructs in which the branch taken is implicitly communicated across the socket by the sending process; this ensures that the two ends cannot get out of step.

Connected session sockets can be passed around between processes, and the SJ system tracks their session types correctly even when they are in mid-communication; this makes it possible to hand off a connected socket to another process to continue the conversation.

5. Two-Way Protocols

We propose generalising occam-π's protocols so that they can specify two-way conversations rather than just sequences of one-way communications.

occam-π's unidirectional protocol specifications can be viewed as a restricted form of session types: all the communications must be in the same direction, only a single choice is permitted at the start of the protocol, and no facilities are provided for iteration and recursion within a protocol (although the same protocol can be used multiple times across the same channel). In order to specify two-way communications, we must relax some of these restrictions.

For example, the **DIE** interface above could be expressed as a single two-way protocol between the client and the server. In this protocol, a client starts a conversation by sending a **roll** or **quit** message; the server will reply to **roll** only with **rolled** or **dropped**. We can specify this as a session type from the client's perspective:

$$(roll! \ . \ INT! \ . \ (rolled? \ . \ INT? \mid dropped?)) \mid quit?$$

(We do not propose that occam-π programmers should use this syntax for protocol definitions – see section 7.)

Protocols can contain multiple direction changes; for example:

$$move! \ . \ (moved? \mid (suspend? \ . \ suspended!))$$

One valid conversation using this protocol would be $move!$, $suspend?$, $suspended!$; another would be $move!$, $moved?$. The conversation $suspended!$ would not be valid.

We constrain the first communication in a two-way protocol specification to always be an output; this makes it possible for the compiler to always be able to tell in which direction the next communication is expected to come.

A client-server connection can now just appear as a channel to the occam-π programmer; there is no need to specify whether a particular communication is a request or a response, since that is implicit in the operation being used. For example, our **DIE** client can now be written this way:

```
PROC roll.die (CHAN DIE die!)
  SEQ
    die ! roll; 6
    die ? CASE
      INT n:
      rolled; n
        ...  rolled an n
      dropped
        ...  dropped the die
:
```

Furthermore, the compiler now has enough information to be able to tell that the following process does not conform to the protocol:

```
SEQ
  die ! roll; 6
  die ! roll; 6
```

Since the session type is now tracked between multiple communications on the same channel, we could allow sequential communications to be split up over multiple communication processes: that is, **die ! roll; 6** would be merely syntactic sugar for:

```
SEQ
  die ! roll
  die ! 6
```

6. Implementation

6.1. Two-Way Channels

Two-way channels could be implemented by the occam-π compiler using a pair of regular channels inside a channel bundle. The transformation required would be very straightforward,

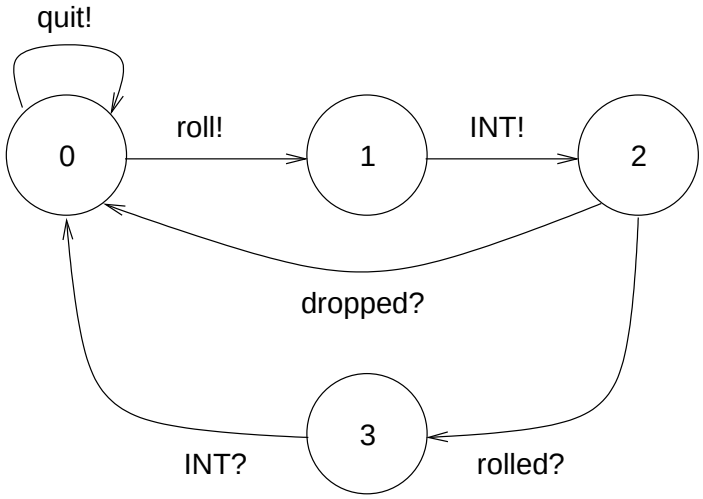

Figure 2. Finite state machine representing the `DIE` protocol

simply selecting the "request" or "response" channel inside the bundle based on the direction of communication. This approach would allow the translation of occam-π code using two-way channels into code that would be accepted by the existing compiler.

However, all existing implementations of occam-π channels on a single machine allow communication in either directions, provided both users of the channel always agree about the direction of communication they expect. Since session types allow the compiler to reason about the direction of communication whenever a channel is used, we can use the existing occam-π runtime's channels as two-way channels with no additional overhead.

This also offers a small memory saving: one channel can now be used where two and a channel bundle were previously necessary. This may be useful in programs with very large numbers of channels.

6.2. Protocol Checking

Checking that one-way protocols are used correctly is simple: each input or output operation must always perform the complete sequence of communications that the protocol describes, so the compiler can tell what communications should happen from the type of the channel alone. Two-way protocols complicate this somewhat because the protocol may take place across multiple operations. We can solve this problem by representing protocols as session types, and attaching a session type to each channel end.

A common representation for a session type is a finite state machine, with each message being an edge in the state machine's graph (see figure 2). The compiler will translate each protocol definition it sees into a state machine; a session type can then be represented as a pair of a state machine and a state identifier within that machine. Given a channel's current state, this makes it possible to tell whether an operation upon it is valid, and if so what the resulting state is. The same approach is already used in Honeysuckle and in implementations of session types in other languages.

Each protocol has an *initial state* for the start of a conversation. Since occam-π protocols may be repeated arbitrarily as a whole, a message with nothing following it in the protocol specification is recorded as a transition back to the initial state.

To check a program for protocol compliance, it is first transformed into a control flow graph (which occam-π compilers already do in order to perform other sorts of static checks). Each channel end variable is tagged with a state, which is set to the initial state when a channel is first allocated. The control flow graph is traversed; when a communication operation is performed upon a channel end, the current state and the message are checked against the

appropriate state machine, and the state is updated. If the communication is not valid for the present state, the compiler can report not only that it's invalid, but also what communications would have been valid at that stage of the protocol.

When two flows of control rejoin, the compiler must check that the state of each channel end is the same in both flows; this ensures that conditionals and loops do not leave channels in an inconsistent state. When a channel is abbreviated – either via an explicit abbreviation, or in a procedure definition – it must be left in the same state at the end of the abbreviation that it was in at the start. (This rule may need to be adjusted to support mid-conversation handoff; see section 7.3.)

Similarly, when a channel end is sent between processes (for example, as part of a channel bundle), its state must be preserved by the communication. As with SJ, it would be perfectly reasonable to hand off a channel end in the middle of a conversation to another process, provided the receiving process agrees what session type it should have. This allows the process network to be dynamically reconfigured without consistency problems.

A **CLAIM** upon a shared channel must start and end with the channel in its initial state, since the channel must be left in a predictable state for its next user. Shared channels are the only case in which channel ends may be aliased in occam-π, so this restriction avoids consistency problems caused by aliasing of session-typed channels.

Note that the channel's state is not tracked at runtime, as with many other session types systems. Two-way protocols incur no runtime overheads.

7. Protocol Specifications

The syntax used so far for session types is hard to read and write, particularly for complex protocols with many choices and direction changes; we would like something more convenient for use in occam-π. We emphasise that we have not yet decided on a final syntax for this; this section describes some of the possibilities we have considered.

7.1. Starting Small

We must preserve the existing syntax for unidirectional protocols in order to avoid breaking existing occam-π code, but since a unidirectional protocol is just a special case of a two-way protocol, we can deprecate the existing syntax in favour of a new one. However, there are some advantages to basing our new syntax on the existing one: the existing syntax has worked well for over twenty years, and occam-π programmers are already familiar with it.

A minimal approach would be to keep the existing syntax for unidirectional protocols – so communications in the same direction are still sequenced using `;` – but allow an indented block inside a protocol specification to mean a change of direction. We could then write our **DIE** protocol as:

```
PROTOCOL DIE
  CASE
    roll; INT
      CASE
        rolled; INT
        dropped
    quit
:
```

This protocol is very simple, but if it had more changes of direction (and therefore deeper nesting), it would be harder to tell the direction of each communication. We could require the user to explicit specify the direction of each communication:

```
PROTOCOL DIE
  CASE
    ! roll; INT
      CASE
        ? rolled; INT
        ? dropped
    ! quit
:
```

The directions are specified from the perspective of the client. This is more useful for documentation purposes; a programmer is more likely to be writing a client to somebody else's server than a server to somebody else's client. Since all the communications within a single **CASE** must be in the same direction, it is somewhat redundant to specify the direction on all of them; it would be possible to apply the direction to the **CASE** itself instead.

This syntax does not allow the full power of session types, though. It is only possible to have choice at the start of a communication or after a change of direction, which means you cannot send some identifying information followed by a command. Furthermore, there is no way to name and reuse parts of the protocol; you cannot write a protocol containing repetition, or share a response (such as a set of error messages) between several possible commands.

7.2. Protocol Inheritance

We have not yet specified how protocol inheritance would work with these simple two-way protocols. We could allow the inclusion of an existing protocol in a new one by giving the existing protocol's name. The effect would be as if the existing protocol's specification were textually included in the new protocol.

```
PROTOCOL ERROR
  CASE
    ! ok
    ! file.not.found
    ! disk.full
:
PROTOCOL FILE
  CASE
    ! open; FILENAME
      ERROR
    ! write; STRING
      ERROR
:
```

Note that the **ERROR** protocol's direction has been implicitly reversed when it is included in **FILE**. In combination with occam-π's existing **RECURSIVE** keyword, which brings a name into scope for its own definition, this approach would allow protocols containing repetition to be defined recursively:

```
RECURSIVE PROTOCOL ARK
  CASE
    ! animal; ANIMAL
      CASE
        ? ok
          ARK
        ? full
    ! done
:
```

This approach has several downsides, though.

It is only possible to recurse back to the "outside" of a protocol. Mutually recursive protocols – which may be useful when you have a protocol that switches between two or more stable states – cannot be written.

It is also difficult to describe the type of a channel in mid-conversation. The session type of a **CHAN FILE** after an **open** or **write** message has been sent is the same: the expected messages are those from **ERROR**. However, it is not a **CHAN ERROR**, because after the error message has been sent the next communication will be one from **FILE**. This makes it impossible to write a reusable error-handling process.

7.3. Named Subprotocols

A more flexible approach would be to allow the user to define named subprotocols within a single protocol definition – which the compiler will eventually translate into named states within the protocol's state machine. Our **FILE** protocol with errors can now be written using a subprotocol for error reporting:

```
PROTOCOL FILE
  SUBPROTOCOL ERROR
    CASE
      ? ok
      ? file.not.found
      ? disk.full
  :

  CASE
    ! open; FILENAME
      ERROR
    ! write; STRING
      ERROR
:
```

Note that the message directions are now written consistently between the top-level protocol and its subprotocol.

We can now refer to a particular state within a protocol when describing a channel's type, which lets us write abbreviations and procedures expecting a channel in a particular state:

```
PROC handle.error (CHAN FILE[ERROR] c?)
  ...
:
SEQ
  c ! open; "foo.occ"
  handle.error (c?)
```

One problem with this is that, by the checking rules described earlier, the abbreviation of **c** in the procedure definition would require it to have the same type when the procedure exited – which will not be the case if it has handled the error. To solve this, we could allow the input and output states of a protocol to be specified in an abbreviation's type – for example, **CHAN FILE[ERROR, FILE] c?**. Another option is to just specify **CHAN FILE** and have the compiler infer the input and output states.

The top-level protocol's name is still made available if **RECURSIVE** is used, so **ARK** can be defined as it is above. We could generalise this to permit mutual recursion between subprotocols, which is rather unusual for occam-π; its scoping rules usually forbid mutual recursion. Mutually-recursive subprotocols would allow us to specify a protocol with multiple "stable

states": for example, a network socket that may be either connected or disconnected, and supports different sorts of requests in different states.

```
PROTOCOL SOCKET
  SUBPROTOCOL DISCONNECTED
    CASE
      ! connect; ADDRESS
        CONNECTED
  :
  SUBPROTOCOL CONNECTED
    CASE
      ! send; DATA
        CONNECTED
      ! disconnect
        DISCONNECTED
  :

  DISCONNECTED
:

CHAN SOCKET c:
...
SEQ i = 0 FOR SIZE addrs
  SEQ
    c ! connect; addrs[i]
    c ! send; "hello"
    c ! disconnect
```

occam-π protocols are not currently written to be this long-lived: a socket protocol in the current language can only describe a single request. Recursive protocols would allow longer-lasting interactions to be captured.

8. Mobile Channels

At present, occam-π allows channel bundles to be mobile, but not individual channels. In order to make a two-way channel mobile, it would need to be wrapped in a channel bundle. It would be more convenient for most uses of two-way channels if the ends of a channel could simply be declared to be mobile in the same way as data and barriers.

The existing syntax for channel end abbreviations in occam-π appends the `!` and `?` decorators to the name of the abbreviation. The same syntax could be used for mobile channel ends:

```
MOBILE CHAN FOO out!:
MOBILE CHAN FOO in?:
SEQ
  out, in := MOBILE CHAN FOO
  PAR
    out ! some.foo
    in ? other.foo
```

However, this makes it impossible to write the type of a channel end on its own – for example, if you wanted to declare a protocol carrying channel ends, or define a type alias. Where does the decorator go? It would be simpler to always include the direction as part of the type of a channel end:

```
MOBILE CHAN! FOO out:
MOBILE CHAN? FOO in:
```

9. Conclusion

We have proposed extensions to the occam-π programming language that would significantly extend the expressive power of channel protocols by permitting two-way communication on a single channel. We have shown how session types can be used to specify these protocols, and to check that processes implement the protocols correctly. We have therefore demonstrated that session types can be used to improve the safety of inter-process communication in an existing concurrent programming language.

Two-way protocols would significantly simplify the implementation of client-server systems in occam-π, but it is important to note that they are not tied to the client-server model. For example, it would be possible to build a ring of processes connected by two-way channels, which would be a violation of the client-server design rules, but permissible using I/O-PAR or other approaches. Two-way protocols simply ensure that the communications between any two connected processes proceed in a consistent manner.

Two-way protocols cannot guarantee that a system follows any particular set of design rules; this is, in general, a difficult problem to solve, particularly in the face of dynamically-reconfigurable process networks. Making guarantees about the system as a whole is out of the scope of this proposal, although we hope that the ability to reason formally about channel protocols using session types would make a future implementation of design-rule checking more straightforward.

Acknowledgements

The author would like to thank Neil Brown and the other members of the Concurrency Research Group at the University of Kent for their work on this proposal.

This work was supported by EPSRC grants EP/P50029X/1 and EP/E053505/1.

References

[1] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4), 1997.

[2] Inmos Limited. occam 2 Reference Manual. Technical report, Inmos Limited, 1988.

[3] Fred Barnes and Peter H. Welch. Prioritised Dynamic Communicating Processes - Part II. In J. Pascoe, R. Loader, and V. Sunderam, editors, *Communicating Process Architectures 2002*, pages 353–370, 2002.

[4] Fred Barnes and Peter H. Welch. Prioritised Dynamic Communicating Processes - Part I. In J. Pascoe, R. Loader, and V. Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, 2002.

[5] Geoff Barrett. occam 3 Reference Manual. Technical report, Inmos Limited, March 1992.

[6] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002. Keynote Tutorial.

[7] Ian R. East. Interfacing with Honeysuckle by Formal Contract. In J.F. Broenink, H.W. Roebbers, J.P.E. Sunter, P.H. Welch, and D.C. Wood, editors, *Communicating Process Architectures 2005*, pages 1–11, September 2005.

[8] Kohei Honda. Types for Dyadic Interaction. In *Proc. CONCUR '93*, number 715 in LNCS, pages 509–523. Springer, 1993.

[9] Matthias Neubauer and Peter Thiemann. An implementation of session types. In Bharat Jayaraman, editor, *PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[10] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. l_{doos}: a Distributed Object-Oriented language with Session types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC 2005*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[11] V. T. Vasconcelos, Antnio Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, number 3170 in LNCS, pages 497–511. Springer-Verlag, 2004.

[12] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Language and Runtime Implementation of Sessions for Java. In Olivier Zendra, Eric Jul, and Michael Cebulla, editors, *ICOOOLPS'2007*, 2007.