Practical Application of CSP and FDR to Software Design

Jonathan Lawrence

IBM United Kingdom Ltd.,
MP 154, IBM Hursley Park,
Winchester, SO21 2JN, UK
jlawrence@uk.ibm.com

Abstract. Most published material on CSP and the FDR tool is theoretical and mathematically rigorous, which can be daunting to the less mathematical software engineer. It is also often difficult to relate the elegant but abstract examples in the literature to the problems of the software engineer who must eventually produce an executable program expressed in a procedural programming language This paper outlines a number of techniques which may be used to model procedural designs in CSP and to structure the refinements so as to render them tractable to verification by the FDR model-checking tool. A simple example, taken from a recent IBM Software Services engagement, is used to illustrate some of the ideas presented in the paper.

1 Introduction

This paper describes some of the author's experiences applying CSP in conjunction with the FDR model-checking tool to a range of small design problems which have arisen in the course of recent IBM Software Services consultancy projects.

1.1 Indebtedness to CSP

The author has been using the CSP notation and FDR tool intermittently for about ten years; initially for the formalization of a concurrent design for the logging component of a transaction processing system, and subsequently for a few other minor pieces of design work and an MSc project.

More recently, and perhaps surprisingly, considerable scope for the application of CSP and FDR has been found in a number of services engagements involving the delivery of bespoke software components or system designs. In all such cases to date, the client has not required and has not been aware that CSP has been used for some aspect of the project; so use of the notation and tools could not be permitted to adversely affect other factors such as performance, function and cost.

That the application of CSP is viable in a commercial environment where cost and delivery schedules are of almost equal importance to quality and reliability, and where neither safety nor security are critical concerns, is a good indication that the combined CSP and FDR approach is sufficiently mature for wider use in software engineering.

A.E. Abdallah, C.B. Jones, and J.W. Sanders (Eds.): CSP25, LNCS 3525, pp. 151 – 174, 2005.

Even in cases where CSP has not been formally used for a design, the conceptual principles behind the notation, and a slightly extended form of CSP communication diagram, have been found helpful in formulating and recording designs.

1.2 Suitability

The CSP approach is most suitable for tackling problems where communication or concurrency is a key concern. In this context, communication includes not only the domain of transport protocols, but also for example a pattern of communication between tightly-coupled components of a software system; while concurrency would include interactions with independent entities such as users and external devices as well as the obvious application to multi-threaded operating environments. It is less well suited to dealing with systems with large and complex state, for which state-based notations such as Z, B or VDM are more appropriate.

An important factor in the successful application of CSP and FDR has been a high degree of selectivity in the choice of problem to tackle. The scope must be sufficiently well-defined to be able to isolate a portion of the system to treat, while being sufficiently complex that there is benefit to be gained from the investment of effort involved. For this reason, the approach has not been applied to every project, and then, typically, only to one aspect of the design.

The remainder of this paper is devoted to an example exemplifying the type of problem to which the approach has been applied, concluding with a summary of the benefits which have been achieved through the use of CSP in software design.

2 Example: A Multi-threaded Connection Pool

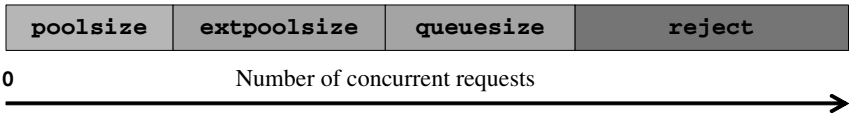
The example presented in this paper was developed as part of a recent IBM Software Services engagement. It illustrates some techniques for the use of CSP and FDR to model and verify software designs; in this case, applied to a multi-threaded connection pooling mechanism forming part of a communications adapter between a Web Server and a transaction processing system.

2.1 Overview

A transport layer to be used for communication provides the notion of a *connection* which may be thought of as an established link between the two systems. Once created, a connection may be used to transmit requests and receive responses on behalf of any client thread; however only one thread at a time may use a connection (this restriction is not policed, but if violated leads to unpredictable results). The creation and destruction of connections is expensive, and the overhead of creating a fresh connection for each client request would be prohibitive. It is therefore necessary to maintain a pool of persistent connections and allocate them to client threads as required, while ensuring that no two threads are ever allocated the same connection concurrently.

Connections are also a limited resource and costly to maintain, so the number of open connections must be carefully controlled, and will usually be less than the number of potential client threads which wish to use them. The design envisaged

allows for a fixed maximum number, `poolsize`, of connections to be permanently allocated; but in order to cater for short-term peaks in demand the system may allocate further connections up to an additional maximum, `extpoolsize`. These extra connections are closed when no longer required. In the event that more concurrent requests are received than can be accommodated within the total, `(maxconn=poolsize+extpoolsize)`, the system may suspend up to `queuesize` threads to wait for a connection to become free; but requests exceeding this limit are rejected. This queueing scheme allows some requests to succeed rather than be rejected, at the cost of some delay, but prevents the system from becoming clogged with suspended threads.



2.2 Specification

Although there is some value in modelling just the design of a software component and then perhaps using a model-checking tool to verify certain desirable properties such as deadlock-freedom; much greater benefit is derived if a specification of the required behaviour is constructed, and the design verified against it. Typically such a specification will be much simpler than the design, such that it can be shown to meet the requirements by inspection or informal arguments, possibly supplemented by additional formal checks using a tool. The level of abstraction to be used in a design is also usually established at the specification stage.

Definitions

Before the specification can be constructed it is necessary to define some datatypes and constants used to label entities and determine system parameters. The datatype `ConnId` introduces a set of tokens used to identify connection instances.

```
datatype ConnId = nil | c1 | c2 | c3
```

`nil` is a special 'null' connection ID which does not refer to a real connection, and is excluded from the set of actual connection IDs.

```
ConnSet = diff(ConnId,{nil})
```

The maximum number of connections which may exist at any time is equal to the number of valid connection IDs, and there must be at least one connection available otherwise all requests will deadlock or be rejected.

```
maxconn = card(ConnSet)
assert maxconn > 0
```

In a multi-threaded design such as this, it is almost always necessary to be able to identify the thread taking part in a particular action, and so the datatype `ThreadId` is defined to provide labels for threads.

```
datatype ThreadId = t1 | t2 | t3 | t4
```

Some constants determining system parameters:

```
poolsize = 2  -- no. of connections to keep open
extpoolsize = maxconn - poolsize  -- extras
queuesize = 2  -- no. of threads allowed to queue
```

All the above size parameters must be non-negative.

```
assert poolsize    >= 0
assert extpoolsize >= 0
assert queuesize   >= 0
```

Possible responses from a call to the pool are defined by the datatype `Response`.

```
datatype Response = ok | error | full
```

`ok` and `error` both indicate that a link request to the target system was made, and it succeeded or failed respectively. `full` indicates that the request was rejected because all available connections were in use, and the queue was full at the time of the request. The level of abstraction to be used in the design is thus already beginning to become apparent from the definition of the possible responses.

Threads

The structure we will use for the specification is a set of independent threads, represented by interleaved processes, handling requests to the system. `call` is the channel on which requests are received at the external interface. `enter` and `exit` are internal channels representing a thread being accepted into, and later leaving the connection pool. All of these carry a label identifying the thread taking part in the event, in order to keep track of which threads are in which state and to tie target links back to the originating thread.

```
channel call,enter,exit:ThreadId
```

Note that no actual request data is represented here, even on the `call` channel. This is a deliberate abstraction from the real system, simply because we do not care about the data for the purposes of this specification – we are solely concerned with the management of the connections, and believe that the transmission of data is a detail which can safely be added at the implementation stage.

The event `reject` is used when a request cannot be processed because the system is full. There is no need to identify a thread on this channel.

```
channel reject
```

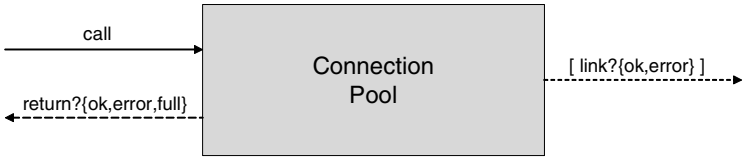
The channel link represents an invocation of the target system. The thread must be identified on this channel since otherwise an implementation would be free to return a response (and probably any associated data) to any thread, rather than the one which made the request.

```
channel link:ThreadId.{ok,error} -- no 'full' on link
channel return:ThreadId.Response -- any poss. Response
```

The process `Thread(t)` models the behaviour of the single thread with label `t`, at the specification level. What we are actually modelling is the behaviour of a thread within the connection pool, which initially only accepts a `call` for that thread, then offers an external choice of `reject` or `enter`. This choice is later resolved by a supervisor process which is monitoring the state of the connection pool.

```
Thread(t) = call.t -> (reject -> return.t.full -> SKIP
                    []  enter.t -> link.t?r ->
                        exit.t -> return.t!r -> SKIP);
                        Thread(t)
```

Depending on the branch chosen, the thread either returns immediately with a `full` response, or issues a `link` to the target system before registering its completion via the `exit` channel and returning control to the caller via `return`. Since the channels `enter`, `exit` and `reject` will be hidden when the specification is assembled, the external view of a thread may be represented by the following diagram:



The diagram represents the connection pool as a black box which accepts `call` requests, may optionally issue a `link`, and then returns to the caller. The response on `return` depends on whether the `link` is issued and if so, what the result was. Threads do not communicate directly with each other, only indirectly through their interactions with shared data or synchronization components; so the combined behaviour of all threads is simply the interleaving of the individual threads.

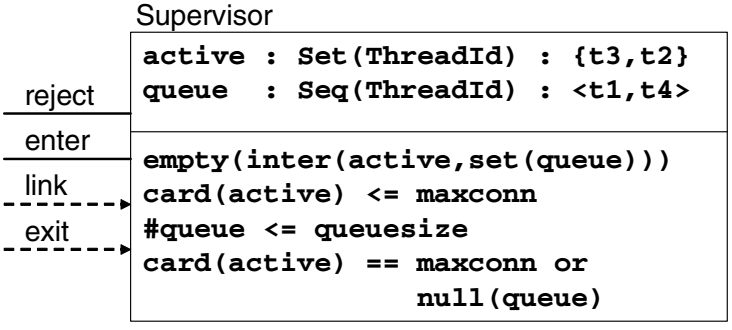
```
Threads = ||| t : ThreadId @ Thread(t)
```

If the internal events were to be hidden at this stage, the external choice on each iteration of a `Thread` would become nondeterministic, and the system would be anarchic, choosing arbitrarily whether to process or reject each request.

Supervisor

To impose order on the system corresponding to our informal requirements for the connection pool, we introduce a supervisor process which maintains a global view of the state of the system, monitoring and controlling the possible actions of the threads according to that view.

The following diagram represents the Supervisor process for the connection pool. Its state comprises two variables: `active`, the set of threads which have been allocated a connection and are in the process of linking to the target system; and `queue`, a sequence of threads which have been accepted but are awaiting the allocation of a connection. Potential example values are given for each variable.



The lower portion of the diagram gives an invariant for the state, which is not necessarily complete, i.e. a partial invariant.

- No thread may be simultaneously active and in the queue.
- The number of active threads is limited to the number of available connections.
- The size of the queue is limited to queuesize.
- All available connections must be in use for a thread to be queued.

The full CSP definition of the Supervisor process is given below. Note that if the stated invariant becomes false after any event, the process deadlocks immediately,

```
Supervisor(active,queue)=
empty(inter(active,set(queue))) and
card(active) <= maxconn and
#queue <= queuesize and
(card(active) == maxconn or null(queue)) &
```

The choice between `enter` and `reject` is based on the state, which is then updated to reflect that choice.

```
(if card(active) < maxconn
 then enter?t -> Supervisor(union(active,{t}),queue)
 else if #queue == queuesize  -- full
      then reject -> Supervisor(active,queue)
      else enter?t -> Supervisor(active,queue^<t>))
[]
```

Threads which are active (i.e. have an allocated connection) are permitted to engage in `link` or `exit`. In the latter case the state is updated as the connection is no longer required by that thread.

```
([]t : active @  -- active threads only
     link.t?_ -> Supervisor(active,queue)
  [] exit.t ->
     let left = diff(active,{t}) within
     if null(queue)
     then Supervisor(left,queue)
     else Supervisor(union(left,{head(queue)}),
                     tail(queue)))
```

The style of expressing a CSP process in the form:

```
P(s) = Inv(s) & [] e : E(s) -> P(s')
```

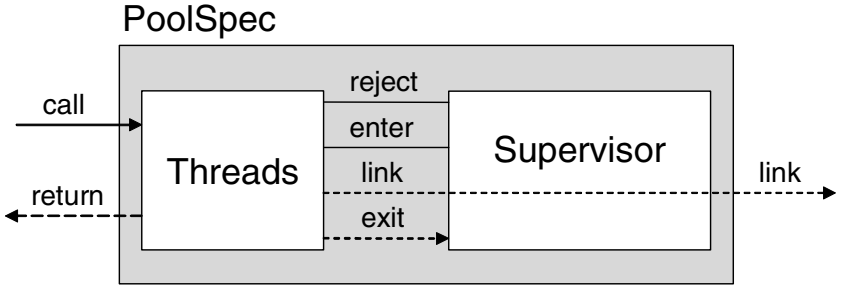
is useful as it allows us to formalize an invariant in a way which will be flagged by FDR if ever violated since it will quickly result in a total deadlock of the whole system. Alternatively divergence could be used in a similar pattern.

Assembly

The complete specification for the connection pool is given by the parallel combination of the threads with the supervisor process in its initial state, synchronizing on the channels shared by the threads and the supervisor, and hiding the internal events.

```
PoolSpec =
    (Threads                              -- all threads
     [|{|enter,exit,reject,link|}|]       -- shared channels
     Supervisor({},<>))                   -- initial state
     \ {|enter,exit,reject|}              -- hide internals
```

The structure of the specification is illustrated in the following diagram.



Validation

At this point, it is appropriate to ask whether what we have specified is actually what we intended, since clearly if it is not, then even a perfect implementation of it will not meet the requirements. There are several techniques which can help to validate the specification:

1. Careful inspection or peer review, paying attention to synchronization and hiding, which are common sources of error.
2. Use of a tool such as ProBe to explore possible behaviour of the specification.
3. Formulation of expected properties of the specification as CSP processes, and then using FDR to check those properties.

A couple of quick checks which require little effort to formulate and are sometimes valid are deadlock and divergence freedom, which both happen to apply in this case. The deadlock freedom check also implicitly checks that the stated invariant for the `Supervisor` state is not violated.

```
assert PoolSpec :[ deadlock free[FD] ]
assert PoolSpec :[ divergence free ]
```

An example of a stronger check of the validity of the specification can be formulated if we consider how we would expect the specification to behave if its `link` channel is hidden. In this case, and abstracting the internal state of the supervisor so that the choice between `reject` and `enter` becomes nondeterministic, each thread may perform an infinite sequence of `call-return` pairs with a nondeterministic choice of response on each `return` event.

```
ThreadInterface(t) = |~| r : Response @
    call.t -> return.t!r -> ThreadInterface(t)
```

The multi-threaded version of the interface should be the interleaving of each thread separately, with no interference between threads. This independence of the threads only holds with `link` hidden since otherwise the refusal of the environment to engage in `link` for one thread may block another waiting in the queue for a connection.

```
PoolInterface = ||| t : ThreadId @ ThreadInterface(t)
```

The FDR refinement check can now be expressed, that our abstract nondeterministic interface specification is refined by `PoolSpec` with the `link` channel hidden. The validity of this assertion in fact includes the deadlock and divergence freedom properties by inspection.

```
assert PoolInterface [FD= PoolSpec \ {|link|}
```

Frequently, the failure of an eventual refinement check of the design will indicate errors or inaccuracies in the specification which need to be corrected. In other words, a ‘correct’ design can be found not to meet the specification originally formulated because the latter is too prescriptive, or some unforeseen subtlety of the operational semantics renders the refinement invalid. In such cases it is the specification rather than the design which needs to be revised and revalidated.

2.3 Design

Design remains the responsibility of the software engineer. CSP and FDR can only help to model, record and verify a design; they cannot help to conceive it. Often, the engineer will have an outline design in mind at the specification stage and this will inform the construction of the specification.

The design for the connection pooling mechanism has four components:

1. The connections provided by the transport layer.
2. A control component which maintains a record of the state of the pool and queue. This is a single shared data component which is used by all threads and which does not provide any synchronization except to protect itself.
3. A dispatcher component which has two functions: synchronization (suspend/resume) of threads in the queue, and connection passing.

4. The threads. Each client thread is a separate process, identical apart from the label used to identify it. These represent independent copies of the same algorithm executing on separate threads, while accessing the same shared components, and as such are similar to the `Thread` processes of the specification.

It would probably be possible to conceive a design (especially in Java) in which the control and dispatcher functions are combined, but separation of these concerns results in a cleaner, more understandable, maintainable and portable structure.

Connections

Although we will not need to implement connections, as they are provided by the transport layer, we need to model them in order to include them in the design. The technique used here is one way to model resources which can be obtained and released, such as memory, objects or in this case, connections. The channels `create` and `close` respectively represent the actions of obtaining and releasing a particular connection, and hence are labelled with a valid connection ID.

```
channel create,close : ConnSet
```

The channel `start_link` is used to represent the use of a particular connection, by a specified thread, to access the target system. The thread ID is necessary for the same reason that it appears on the `link` channel used in the specification; indeed, `end_link` will later be renamed to `link` when the design is fully assembled.

```
channel start_link    : ConnSet.ThreadId
channel end_link      : ConnSet.ThreadId.{ok,error}
```

The link channel of the specification has been split into two separate channels for the design. This allows the model to include the possibility of interleaving of link requests which we wish to guard against, so that its occurrence can be detected by FDR. The complete interface of the transport layer is given by the following set definition and will be useful later.

```
ConnInterface = {|create,close,start_link,end_link|}
```

It is useful to define a divergent process which can be used to represent a broken component – often one which has been used in some invalid way. If this state is reached in an FDR check of the design it will cause the check to fail. A simple divergent process is:

```
DIV = STOP |~| DIV
```

Before a connection has been created, or while a `link` request is being processed by a connection on behalf of a thread, it should be invalid for any `close` or `start_link` event to occur for a connection. `ConnError` is a process which may always accept any such event and then immediately diverge, causing FDR to flag its occurrence during a refinement check.

```
ConnError(c) = [] e : {|close.c,start_link.c|} @
                    e -> DIV
```

Initially, a connection may be considered to be in a latent, unobtained state in which it can only validly engage in the action of being created.

```
Connection(c) = create.c -> Active(c)
              [] ConnError(c)
```

Any attempt to close or use a connection before it has been created will result in divergence, so if our design does this it will be detected by FDR. After creation, a connection may be closed, or used by any thread to initiate a `link` to the target system. In the latter case it moves to a distinct `Linking` state.

```
Active(c) = close.c -> Connection(c)
          [] start_link.c?t -> Linking(c,t)
```

In the `Linking` state a connection may complete the `link` request and return to the `Active` state ready for other requests, but we also allow the possibility of an invalid event (`close` or `start_link`), leading to divergence. If this scenario can arise in the assembled design it will be detected by FDR, allowing us to police the requirement that link requests are not interleaved on a connection. This could not be done if a `link` remained as a single atomic event in the design.

```
Linking(c,t) = end_link.c.t?_ -> Active(c)
             [] ConnError(c)
```

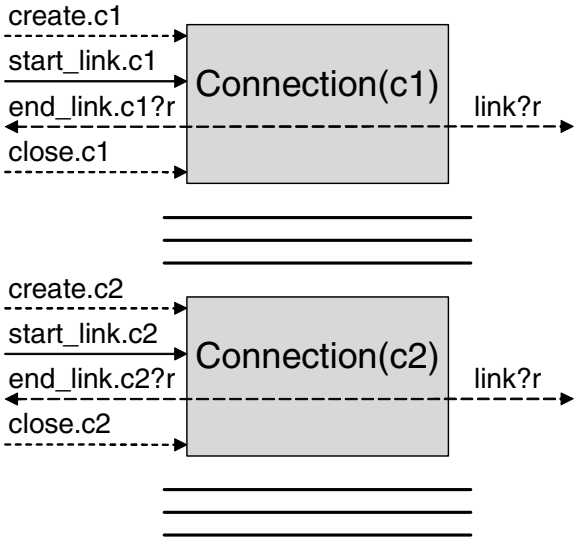
In fact, although a connection is modelled as remaining active after `end_link` we will regard an `error` response as indicating a possible problem with that connection, and close it without further reuse. Connections are independent of each other, so the complete transport layer is represented by the interleaving of all possible valid connections.

```
Connections = ||| c : ConnSet @ Connection(c)
```

The following diagram illustrates the structure of `Connections`, showing how it is composed of the interleaving of several independent `Connection` processes labelled by unique connection IDs. The meaning of the arrows, dashed and dotted lines for the channels is as explained earlier for the high-level description of the Connection Pool.

The diagram also anticipates the eventual renaming of `end_link` to `link` in the final system, to conform to the external interface of the specification. This is explained further when the complete system is assembled later.

This simple model of the transport layer relies for its validity on the way it is used by the threads: when creating a connection a thread must use an external choice over all valid connection IDs (… `-> create?c ->` …) – it may not attempt to create a particular connection although this is not prohibited by the model.



Control

The `Control` component keeps track of the state of the pool. It provides two functions: 1) obtain a connection from the pool; and 2) return a connection to the pool after use. Often when modelling a shared data component such as this, each function will be represented by a single channel, but in this case a more complicated pattern is used where each function can result in a choice over several channels, this choice being determined by the values of the state variables.

The following channels (plus `reject` which has already been defined) are used to request a connection from the `Control` component. The client thread must offer an external choice of these channels when requesting a connection from `Control`, and subsequently act according to the channel actually chosen.

```
channel allocate,reject  -- create a new connection
channel reuse : ConnSet  -- reuse a pooled connection
channel wait : ThreadId  -- suspend thread
```

In a similar way, the following channels are used to return a connection to the pool:

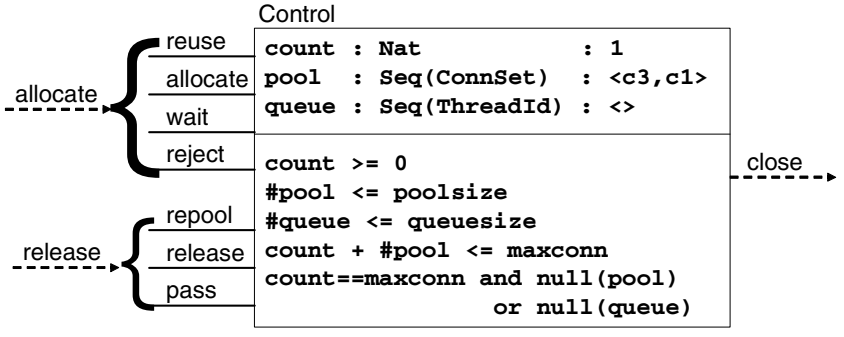
```
channel repool:ConnId -- this may be nil indicating
that the connection is closed
channel release -- release the connection if not
already closed
channel pass:ThreadId -- pass the connection to this
thread
```

It is convenient, and reduces the likelihood of errors when assembling the complete system, to define the set of all events in the client interface of `Control`:

```
ControlInterface = {|reject,allocate,reuse,wait,
                     repool,release,pass|}
```

The channel `close` is not included in this definition as it is a demonic event of the component and is not synchronized with the client threads.

`Control` may be represented by the following diagram, following a similar pattern to that for the `Supervisor` process of the specification:



As for `Supervisor`, potential example values are given for each state variable, and an invariant is specified in the lower portion of the diagram, which as before, may be only partially complete.

The complete CSP definition for the Control component follows.

```
Control(count,pool,queue) =
count >= 0                  and
#pool <= poolsize           and
#queue <= queuesize         and
count + #pool <= maxconn    and
((count == maxconn and null(pool)) or null(queue)) &
((not null(pool) & let front^<last> = pool within
  STOP |~| close!last -> Control(count,front,queue))
[]          -- cases when requesting a connection
( if count==maxconn
  then if #queue < queuesize
       then wait?t -> Control(count,pool,queue^<t>)
       else reject -> Control(count,pool,queue)
  else if null(pool)
       then allocate -> Control(count+1,pool,queue)
       else reuse!head(pool) ->
            Control(count+1,tail(pool),queue) )
[]        -- cases when returning a connection
( if null(queue)
  then if #pool == poolsize
       then release -> Control(count-1,pool,queue)
       else repool?c ->
            if c == nil
            then Control(count-1,pool,queue)
            else Control(count-1,<c>^pool,queue)
  else pass!head(queue) ->
       Control(count,pool,tail(queue))))
```

Dispatcher

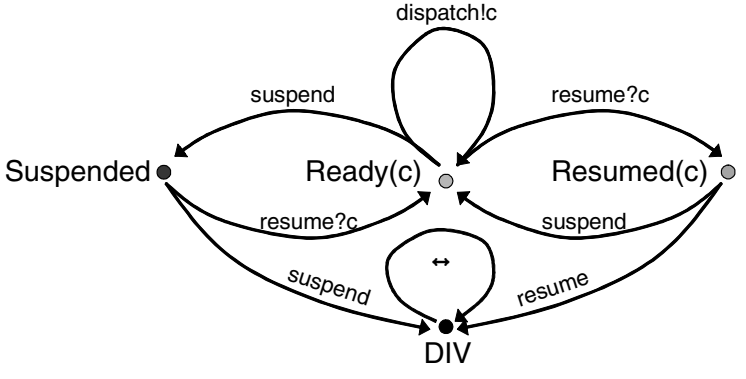
The dispatcher component combines two functions: explicit thread synchronization (suspend / resume) and connection passing. The suspend channel identifies the thread which is to be suspended:

```
channel suspend : ThreadId
```

The channels `resume` and `dispatch` each identify the thread being resumed, but also transmit a connection to be used by that thread. This is an input on the resume channel, and an output on dispatch.

```
channel resume,dispatch : ThreadId.ConnId
Ready(t,c) = dispatch.t.c -> Ready(t,c)
             [] suspend.t -> Suspended(t)
             [] resume.t?c -> Resumed(t,c)
Suspended(t) = resume.t?c -> Ready(t,c)
            [] suspend.t -> DIV
Resumed(t,c) = suspend.t -> Ready(t,c)
            [] resume.t?_ -> DIV
```

A state transition diagram illustrating the Ready process for a thread is given below.



The complete Dispatcher function for all threads is simply the interleaving of each individual thread's dispatcher, initialized with a null connection.

```
Dispatcher = ||| t : ThreadId @ Ready(t,nil)
```

As with Control it is convenient to define the set of all events in the interface of Dispatcher.

```
DispatchInterface = {|suspend,resume,dispatch|}
```

Threads

We now model the actions of a client thread interacting with `Control`, `Dispatcher` and `Connections`. We call this `Client(t)`, labelled by `ThreadId`. Essentially this

is a CSP model of the algorithm to be followed by a thread invoking the call function of the connection pool. There are 3 stages to the processing:

1. Obtain a connection from the pool if possible, or wait for one to become available (if neither of these is possible then the request is rejected).
2. Use the connection obtained in stage 1 to link to the target system.
3. Release the connection to the pool and return to the caller.

As in the specification, each thread is modelled as a separate copy of identical processes with events labelled with a `ThreadId`.

```
Client(t) = call.t ->       -- caller initiates process
(  reject -> return.t!full -> Client(t)
[] wait.t -> suspend.t -> dispatch.t?c ->   -- suspend
             (if c != nil then Execute(t,c)  -- valid
              else create?d -> Execute(t,d)) -- need new
[] allocate -> create?d -> Execute(t,d)     -- need new
[] reuse?c -> Execute(t,c) )    -- connection from pool
```

The process `Execute(t,c)` represents thread `t` once it has obtained a valid connection `c`. It initiates a link to the target system via `start_link` using the connection, and then waits to engage in the corresponding `end_link` event.

```
Execute(t,c) =  -- thread has valid connection c to use
start_link.c.t -> end_link.c.t?r -> -- link to target
if r == ok then Release(t,c,r)    -- retain connection
else close.c -> Release(t,nil,r)  -- close due to error
```

Finally the thread must release the connection back to `Control` and return to its caller. As when obtaining a connection, the thread must offer an external choice over the `Control` channels used for release, and take appropriate action depending on the channel chosen by `Control`. In all cases, the last event is to return the response from the link, before reverting to the initial state to await the next call on that thread.

```
Release(t,c,r) =
 ( repool!c -> SKIP -- c is back in pool, no more to do
[] release -> (if c==nil then SKIP    -- already closed
               else close.c -> SKIP)  -- must close c
[] pass?u -> resume.u!c -> SKIP); -- pass c to thread u
   return.t!r -> Client(t) -- return response to caller
```

Threads do not communicate except indirectly through their interactions with the shared components, so the combined behaviour of all the client threads is simply the interleaving of the individual threads.

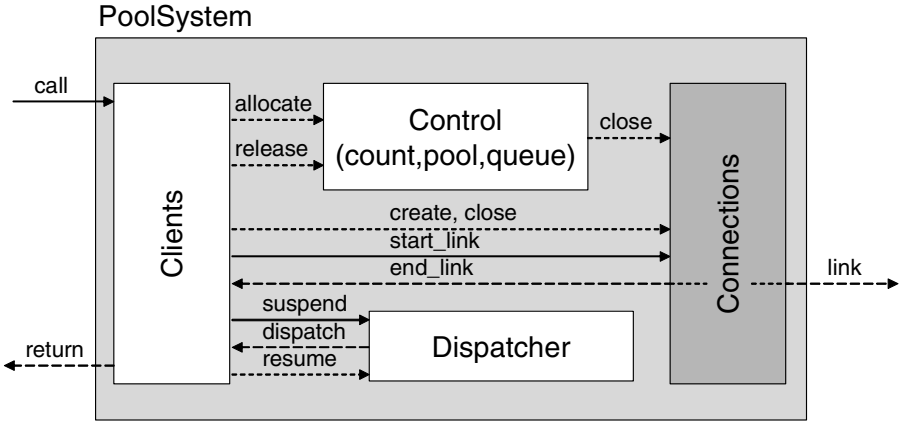
```
Clients = ||| t : ThreadId @ Client(t)
```

Assembly

The following diagram illustrates how the complete CSP model of the implementation is assembled from its component parts.

Starting with `Clients`, we add the other components one at a time, synchronizing on the shared interface events and then hiding them at each stage. `DispatchClients` is the combination of `Clients` with `Dispatcher`.

```
DispatchClients = ( Clients
                    [|DispatchInterface|]
                    Dispatcher ) \
                    DispatchInterface
```



Next, we add the `Control` component. Note that because `close` is not in `ControlInterface`, `close` events from `Clients` and from `Control` are interleaved rather than synchronized and so may occur independently.

```
ControlClients = ( DispatchClients
                   [|ControlInterface|]
                   Control(0,<>,<>) ) \
                   ControlInterface
```

Next, the `Connections` are added in a similar way. The `end_link` channel remains exposed at this stage as it will become the `link` channel from the specification.

```
ConnClients = ( ControlClients
                [|ConnInterface|]
                Connections ) \
                {|create,close,start_link|}
```

Finally, all `end_link` events are renamed to `link` for compatibility with the specification. This involves removing the `ConnId` labels from the events as they are no longer relevant.

```
PoolSystem = ConnClients [[end_link.c <- link |
                                    c <- ConnSet]]
```

Verification

The required refinement relationship between the specification and design is expressed as an FDR assertion using the most general semantic model.

```
assert PoolSpec [FD= PoolSystem
```

We are now ready to perform the verification using FDR. The structure of the system means that any check of the above assertion implicitly subsumes the corresponding checks for any subset of `ThreadId`; however it is still useful to start with one thread and work up as this will detect errors which become apparent only when a certain number of threads access the system concurrently, as well as ensuring that we begin well within the capacity of the tool.

For all combinations of system parameters which have been tried and for which the check completed, the assertion holds. Typical output from the end of a check is given below, in this case for four threads, three available connections, a `poolsize` of 2 and a `queuesize` of 1.

```
. . .
+.41,850,000 *
+.*
+.*
+.... 41,855,808
Refine checked 41,855,808 states
With 198584320 transitions
Took 7166(6791+57) seconds
```

The following table gives the corresponding number of states and transitions for a few different checks, which are of interest if only to show how the size of the check depends on the number of threads and other parameters.

#t	#c	pool size	queue size	States	transitions
3	1	0	0	1,847	5,916
3	1	1	1	48,392	146,032
3	2	0	1	87,708	278,538
3	2	1	1	110,712	376,926
4	2	2	1	54,781,182	241,887,276
4	3	2	1	41,855,808	198,584,320

2.4 Implementation

This section illustrates how a CSP design such as our connection pool can be recast into an executable procedural program, in this case in the Java language. The translation process is manual, and involves not only a change of language, but also the removal of abstractions present in the design and some minor enhancements not reflected in the CSP model.

The Java code presented here is a simplified version of the actual implementation from the project, intended to make the relationship to the CSP design clearer.

Dispatcher

The `Dispatcher` class implements the `Ready` process for a thread, so there will be a separate instance of this class for each thread in the queue. The methods of this class, `suspend()` and `resume()`, are both `synchronized` (in the Java sense), so that their actions are effectively atomic as in the CSP model, as well as being necessary in order to use the Java `wait()/notify()` mechanism.

constants

Three constants are defined corresponding to the possible states of the dispatcher as follows.

```
private static final int ready     = 1; -- Ready(c)
private static final int suspended = 2; -- Suspended
private static final int resumed   = 3; -- Resumed(c)
```

variables

Each `Dispatcher` instance has two variables; `state`, which takes one of the three constant values defined above, and `conn`, which is the Connection passed to the thread by `resume()`. The initial values of the variables correspond to the process `Ready(t,nil)` as in the CSP definition of the `Dispatcher` component.

```
private int        state = ready;
private Connection conn  = null;
```

suspend()

The action of `suspend()` depends on the state at the time it is invoked. If `ready`, it is moved to the `suspended` state and caused to wait for the corresponding call to `resume()`. If already resumed, there is no need to wait and the thread can proceed immediately. The connection stored on `resume()` is returned to the caller.

```
synchronized Connection suspend() { // suspend.t ->
  switch (state) {
    case ready :
      state = suspended;
      try { wait(); }
      catch (InterruptedException e) {}
      break;
    case suspended :
      exception("Already suspended");
    case resumed :
      state = ready;
      break;
  } // switch()
  return conn;  // -> dispatch.t!conn
} // suspend()  // -> Ready(conn)
```

resume()

The `resume()` method stores the connection being passed to the target thread in the instance and then modifies the state depending on its initial value in accordance with the CSP. If a thread is already suspended, then it is notified to allow its `wait()` to complete.

```
synchronized void resume(Connection c) {
  conn = c;
  switch (state) {
    case ready :
      state = resumed;
      break;
    case suspended :
      state = ready;
      notify();
      break;
    case resumed :
      exception("Already resumed");
  } // switch()
} // resume()
```

Pool

The Java class `Pool` combines the implementations of two components of the design: `Control` and `Client(t)`. Roughly speaking, the instance variables together with the `synchronized()` blocks within the `allocate()` and `release()` methods correspond to `Control`; whilst `call()` and the remaining code from the other methods together implement `Client(t)`. The melding of the two CSP processes is an implementation convenience partly due to the fact that Java allows only a single return parameter on a method call. Note that the thread executing this code is never identified explicitly as it is in the CSP but is always present by implication.

constants

Three constants are defined corresponding to the CSP datatype `Response`.

```
public static final int ok    = 1;
public static final int error = 2;
public static final int full  = 3;
```

variables

The instance variables of `Pool` have an obvious correspondence with the state variables of the CSP `Control` process in the design. `count` is a simple integer, whilst the two sequences `pool` and `queue` are each implemented by a Java `Vector` object. The initial values of these variables correspond to the initial state of `Control` in the assembled `PoolSystem`, i.e. `Control(0,<>,<>)`.

```
private int    count = 0;
private Vector pool  = new Vector();
private Vector queue = new Vector();
```

allocate()

The `synchronized()` block in this method implements the CSP choice between the possible connection allocation events of `Control`. The choice which is made is communicated to the subsequent code (part of `Client(t)`) by different combinations of local variables.

```
private Connection allocate() {
  Dispatcher thread = null;
  Connection conn = null;  // allocate
  boolean queue = false;
  synchronized (this) {
    if (count >= maxconn) {
      if (queue.size() < queuesize) {
        thread = new Dispatcher();
        queue.add(thread);
        queue = true; // wait
      }
      else return null; // reject
    }
    else {
      if (!pool.isEmpty()) // reuse
        conn = pool.remove(pool.size()-1);
      count++;
    }
  } // synchronized()
  if (queue) conn = thread.suspend();
  if (conn == null) conn = create();
  return conn;
} // allocate()
```

release()

As with allocate(), the synchronized() block here implements the CSP choice between the possible connection release events of Control. The choice is communicated to the subsequent code by different combinations of the local variables waiter and conn, which then behaves according to the corresponding path of the Client(t) CSP process.

```
private void release(Connection conn) {
  Dispatcher waiter = null;
  synchronized (this) {
    if (queue.isEmpty()) {
      count--;
      if (conn != null) pool.add(conn); // repool
      if (pool.size() > poolsize)
        conn = pool.remove(0); // release
      else conn = null;
    }
```

```
      else waiter = queue.remove(0); // pass
    } // synchronized()
    if (waiter != null) waiter.resume(conn);
    else if (conn != null) conn.close();
  } // release()
```

call()
This is the only public method of `Pool`, and implements those sections of the `Client(t)` process not included within the `allocate()` or `release()` methods, including the top-level `call` and `return` events. Note the introduction here of data to be exchanged with the target system on `link()`. The omission of this data from the CSP model is one of the abstractions employed in the design.

```
public int call(byte[] data) {
  Connection conn = allocate();
  if (conn==null) return full; // return.t!full -> …
  else { // Execute(t,conn)
    boolean success = conn.link(data);  // ok | error
    if (!success) { // r != ok
      conn.close;   // close.conn ->
      conn = null;  // Release(t,nil,r)
    }
    release(conn);  // Release(t,conn,r)
    if (success) return ok;  // return.t!r -> Client(t)
    else return error;
  } // else()
} // call()
```

3 Summary

The original design and implementation of the connection pooling component described in this paper was completed in three days, from the preliminary CSP specification to initial testing of the Java code, including verification of the design using FDR. This time was split approximately equally between developing and verifying the CSP design, and recasting it as executable Java. Following delivery of the system containing the connection pooling mechanism to the client shortly thereafter, no errors have been detected in the implementation in spite of thorough testing and heavy usage of the system by the client.

The implementation was subsequently enhanced with some functions not included in the CSP model, notably the ability to cause threads which have been queueing for more than a certain interval to time out. These modifications were not added to the CSP model (although it would have been perfectly feasible to do so), because it was thought that the effort involved would not be justified by the likely benefit in

verifying the enhancements. Rather it was considered that the clear structure and design intent engendered by the original use of CSP meant that the necessary modifications could be made without risk to the integrity of the core design; and this appears to have been borne out in practice.

Other Techniques

The example in this paper has illustrated some techniques for modelling procedural, and in particular multi-threaded, designs in CSP such that they may be checked by the FDR tool. For example:

- Abstraction.
- Specification; and validation of specification properties.
- Modelling multiple threads including non-interleaving properties.
- Design – decomposition into data / synchronization / processing elements.
- Resource allocation and deallocation.
- Specifying and checking state invariants of processes.
- Design verification with FDR refinement assertions.
- Implementation by translation of CSP to procedural code.

A single example, however, can only exemplify a small cross-section of the techniques which might need to be applied to model and verify a wider range of problems. In particular the example used in the paper is sufficiently simple that one stage of refinement is sufficient to reach an (almost) directly implementable level of design. This is by no means always the case and several techniques may need to be applied to deal with larger problems. Some of these are summarized below.

- Stepwise refinement. A crucial property of CSP semantics is that all CSP operators are monotonic with respect to refinement. This allows an abstract or not directly implementable process to appear at one level of a design, and for it to be refined and checked separately. An unmanageably large design may thereby be broken down into several more manageable design steps, each independently verifiable by FDR.
- Interface wrapping. Often, where stepwise refinement is used, the intermediate abstract component may not be directly refineable because its interface refers directly to events which will not exist or will not be exposed in the design. In such cases an additional call-return interface layer may be inserted to encapsulate the component and the wrapped version then refined. The validity of the introduction of this additional interface layer may itself be checked, often at a single thread level.
- Interface protocols and rely-guarantee contracts. A ‘correct’ design may not be a true CSP refinement of the specification because of some reliance on the way the system will be used. It is usually possible to deal with such cases by formalizing the permissible usage scenarios as a CSP process and including this in parallel with the specification to be refined.
- Avoidance of unbounded state. The FDR tool is not able to check systems with unbounded or even very large state spaces and there are several ways of reducing or avoiding such problems; for example:

  - Factor out unbounded state components from the system.
  - Use modulo arithmetic to reduce state space of numeric types.
  - Place bounds on counts by introducing artificial deadlock or divergence in a specification.
- Data independence. This idea has already been mentioned elsewhere and involves replacing a large datatype with a much smaller one for the purpose of the model.

Even for the example in the paper, the step from the CSP `Control` component of the design to the implementation of the `allocate()` and `release()` methods of the Java `Pool` class is not entirely obvious, and an additional stage employing interface wrapping and stepwise refinement might have been added.

A couple of other techniques which can be useful for certain special classes of problem are:

- Discrete time modelling. FDR does not include support for the semantics of Timed CSP, however some timing aspects can be modelled and checked by FDR using a technique of 'untimed time', in which the 'tocks' of a clock are represented as CSP events in the untimed language.
- Fairness modelling. CSP does not have any built-in notion of fairness, in other words there is nothing in the language to prevent infinite overtaking from occurring. However, it is perfectly feasible to construct an explicit representation of a concept of fairness for any given system. The form of this representation is typically system dependent but can be similar to the way that lossy channels are sometimes represented in communication protocols.

4 Conclusion

This paper has presented one example of the application of an approach to software development involving the CSP notation for modelling combined with the use of the FDR model-checking tool for validation and verification of the specification and design respectively.

Benefits

Apart from the obvious benefit of the capability for automated verification of designs from their specifications, the use of CSP in software engineering has other advantages.

- Discipline for structuring designs. The use of CSP naturally encourages the decomposition of a design into clearly defined logical units, resulting in a more understandable and maintainable implementation structure.
- Elegance and efficiency. In the author's opinion, the use of CSP tends to result in designs which are more elegant and economical, both in terms of the amount of code required and its runtime efficiency.
- Design documentation. In common with other design methodologies, the CSP approach inevitably results in the production of design documentation at a higher level of abstraction than the eventual code. Where connection diagrams

are used, the diagrams record the overall structure, whereas the details of the behaviour of individual components are in the CSP.
- Hierarchical decomposition. In larger systems, a natural consequence of stepwise refinement is that the design is split up into manageable chunks which can be understood and implemented largely independently of each other.

Limitations

Probably the main limitation of the approach discussed in the paper is the restriction on the size of system which can be checked by the tool. For example, the example in this paper can be checked with up to four threads and any given combination of the other parameters in the space of a few hours on a modern workstation. However, the state space which needs to be explored increases approximately exponentially with the number of threads and when an additional thread is defined the check can only be completed for a small subset of combinations of the other parameters.

Consequently, verification of such a design by FDR for certain specific cases can provide a considerable level of confidence in the correctness of a design, but cannot prove it to be correct as the system is scaled up. An exception is where data-independence is exploited, as this is known to scale up without affecting the validity of checks performed with small datatypes. For certain very restricted classes of problem the scalability limitation can be overcome by an inductive technique but this is not applicable to designs such as that presented in this paper.

Other Examples

Some other recent examples of the use of the combined CSP and FDR approach from the author's consultancy work include:

- A design for a concurrent twin-buffering logger.
- A transport protocol for transmitting 'unbounded' data in finite segments.
- A design for the Web-enablement of a CICS 3270 application.
- A mechanism to ensure once-only initialization under race conditions.
- A model of a bimodal locking algorithm for Java objects.

These have mostly been of a similar size and complexity to the example in this paper and the results, in terms of the benefit from the application of the approach have also been comparable. A few other examples where adapted connection diagrams only have been applied to formulate and document a design are:

- A CICS TCP/IP socket listener-server.
- A framework to demonstrate tightly-coupled transactional interoperation between independent Java and C applications.
- Control flow in an XML reformatting tool.

In these and other cases not mentioned, the diagrams have proved beneficial in imposing a discipline of decomposing a system into logically organized components, defining the possible interactions between them, and subsequently providing a record of the design to aid in the construction of the system.

It is also worth mentioning that the part-time MSc in software engineering run by Oxford University Computing Laboratory has proved extremely popular with Hursley employees, several of whom have chosen to undertake projects using CSP and FDR

for the dissertation element of their studies. These projects are normally based on an aspect of the student's work responsibilities.

Future Outlook

The software engineering community would undoubtedly benefit from a wider knowledge and application of CSP and CSP-based model-checking tools, such as FDR, even in areas where safety and reliability are not overriding priorities. Developments which might help to facilitate this would be:

1. Free availability of FDR, preferably as an open-source project.
2. Availability of a *practical* manual on the use of CSP and associated tools for software modelling, design and verification.
3. Inclusion of CSP connection diagrams in UML, perhaps with some extensions such as those employed in this paper.

CSP is an extremely powerful language for specification and modelling the design of software, especially system components in which communication or concurrency are central issues. Used in conjunction with a model-checking tool such as FDR, the notation provides unparalleled capability for the automated checking of designs which would otherwise be extremely difficult to verify.

Bibliography

- *Communicating Sequential Processes*, C.A.R. Hoare, Prentice-Hall International, 1985.
- *The Theory and Practice of Concurrency*, A.W. Roscoe, Prentice Hall, 1998.
- *Failures-Divergences Refinement*, FDR2 User Manual, Formal Systems (Europe) Ltd., 1998. (www.fsel.com).

Trademarks