Verification of a Dynamic Channel Model using the SPIN Model Checker

Rune Møllegaard FRIBORG [a,1] and Brian VINTER [b]
[a] *eScience Center, University of Copenhagen*
[b] *Niels Bohr Institute, University of Copenhagen*

Abstract. This paper presents the central elements of a new dynamic channel leading towards a flexible CSP design suited for high-level languages. This channel is separated into three models: a shared-memory channel, a distributed channel and a dynamic synchronisation layer. The models are described such that they may function as a basis for implementing a CSP library, though many of the common features known in available CSP libraries have been excluded from the models. The SPIN model checker has been used to check for the presence of deadlocks, livelocks, starvation, race conditions and correct channel communication behaviour. The three models are separately verified for a variety of different process configurations. This verification is performed automatically by doing an exhaustive verification of all possible transitions using SPIN. The joint result of the models is a single dynamic channel type which supports both local and distributed any-to-any communication. This model has not been verified and the large state-space may make it unsuited for exhaustive verification using a model checker. An implementation of the dynamic channel will be able to change the internal synchronisation mechanisms on-the-fly, depending on the number of channel-ends connected or their location.

Keywords. CSP, PyCSP, Distributed Computing, Promela, SPIN.

Introduction

Most middleware designers experience situations where they need to choose between generality and performance. To most experienced programmers this dilemma is natural since high performance implementations are typically based on assumptions that from the usage point translates into limitations. The PyCSP project has since the beginning had a strict focus on generality, attempting to present the programmer with only one channel type and one process type. The single channel type has succeeded while the single process type has been attempted through individual PyCSP packages with separate process types. These process types are: greenlets (co-routines), threads and processes, as seen from the operating system view. The reason for the three different packages were to enable PyCSP applications to have up to 100,000 CSP processes (co-routines), posix threads for cross-platform support and OS processes executing in parallel while not being limited by the CPython Global Interpreter Lock [1].

We would like to reach the point of only one process type and one channel type, but for this to be possible we need a channel type which preserves the qualities of the previous channel type, i.e. be of the kind any-to-any and support external choice in both directions as well as offer both channel poisoning and retirement like the existing PyCSP channels. The new channel must support three possible process locations: within the same PyCSP process,

[1]Corresponding Author: *Rune Møllegaard Friborg, eScience Center, University of Copenhagen, DK-2100 Copenhagen, Denmark*. Tel.: +45 3532 1421; Fax: +45 3521 1401; E-mail: `runef@diku.dk`.

on a different PyCSP process within the same compute-node and, finally, a different PyCSP process on a different compute-node. It is of course trivial to make a common channel type based on the lowest common denominator, i.e. a networked channel, since this will also function within a compute-node and even within a process. However, the downside is self-evident since the overhead of deploying an algorithm based on shared-nothing mechanisms is much slower than algorithms that can employ shared state.

The solution to the proposed problem is a channel that can start out with the strongest possible requirements, i.e. exist within a single process, and then dynamically decrease the requirements as needed, while at the same time employing more complex, and more costly, algorithms.

The present work is a presentation of such a dynamic channel type. The algorithms that are employed grow quite complex to ensure maximum performance in any given scenario, thus much work has been put into verifying the different levels a channel may reach. Using the SPIN model checker, we perform an exhaustive verification of the local and distributed channel levels for a closed set of process configurations. These include any-to-any channels with input and output guards in six different combinations.

Background

PyCSP is currently a mix of four implementations providing one shared API, such that the user can swap between them manually. The four implementations do not share any code and the different channel implementations can not function as guards for a single external choice, since they are not compatible.

Listing 1. A simple PyCSP example demonstrating the concurrent nature in CSP upholding to unbounded non-determinism and protected against race conditions during termination.

```
# Select implementation
import pycsp.threads as pycsp
# (alternatives: pycsp.processes, pycsp.greenlets, pycsp.net)

@pycsp.process
def source(chan_out, N):
    for i in range(N):
        chan_out("Hello (%d)\n" % (i))
    pycsp.retire(chan_out) # The channel has one less writer

@pycsp.process
def sink(chan_in):
    # The loop terminates on the signal that announces that all
    # writers have retired
    while True:
        sys.stdout.write(chan_in())

ch = pycsp.Channel()
pycsp.Parallel(                    # Run in parallel
    5 * source(ch.writer(), 10),   # Five source processes
    5 * sink(ch.reader())          # Five sink processes
)
```

In listing 1, we show a simple application using the current PyCSP where channels are any-to-any, synchronous and uni-directional. Processes can commit to reading or writing from single channels or they can commit to a set of distinct channels using the alt (external choice) construct. Committing to the alt construct means that exactly one of the channel

operations will be accepted and all others are ignored. The alt construct allow a mix of read and write operations. The first model, which we present in section 2.1, is a model of the current channel implementation for threads in PyCSP.

In [2] we presented PyCSP for scientific users as a means of creating scalable scientific software. The users of a CSP library should not have to think about whether they might be sending a channel-end to a process that might be running in a remote location. Or how they work around an external choice on channels, that does not support it. One of the powerful characteristics of CSP is that every process is isolated, which means that we can move it anywhere and as long as the channels still work, the process will execute. Because of this, processes can easily be reused, since all the inputs and outputs are known.

The network-enabled PyCSP implementation is a prototype and uses a single channel server to handle all channel traffic. The single channel server runs the thread implementation of PyCSP internally, which creates a temporary thread for every request. The server is a serious bottle-neck for the channel communication and has limited the type of parallel applications implemented in PyCSP. In this paper we present a distributed channel model and check its correctness using the SPIN Model Checker.

Promela and the SPIN Model Checker

Promela (Process Meta Language) is a process modeling language whose purpose is to verify the logic in concurrent systems. In Promela models, processes can by created dynamically and can communicate through synchronous or asynchronous message channels. Also the processes are free to communicate through shared memory. If variables or channels are created globally, then they are available through shared memory to all Promela processes. Promela has a basic set of types for variables: bit, bool, byte, mtype (similar to enum), short and int. The models presented in this paper use shared memory when modeling internal communication, and message channels when modeling distributed communication.

In Promela, every statement is evaluated to one of two states, it is either *enabled* or *blocked*. Statements as assignments, declarations, **skip** or **break** are always enabled, while conditions evaluated to false are blocked. When a statement is blocked, the execution for that process halts until the statement becomes enabled. The following is an example of a blocked statement following an enabled statement:

```
value = 1;        /* enabled */
value == 0;       /* blocked */
```

When executing (simulating) a Promela model, the statements in concurrent processes are selected randomly to simulate a concurrent environment. To allow modeling synchronisation mechanisms, a sequence of statements can be indicated as atomic, by using the **atomic** keyword and enclosing the statements in curly brackets:

```
atomic {
   /* The first statement in an atomic region is allowed to block. */
   process_lock == 0;
   process_lock = 1;
}
```

To organise the code in Promela we use inline functions. When declaring inline functions, the parameters in the parameter list have no types. The inline functions are exclusively used as a replace-pattern, when generating the complete model with a single body for each thread. All values passed to an inline function is pass-by-reference. There is no return construct in Promela, thus values must be returned by updating variables through the parameter list.

Control flows in Promela can be defined using either **if** .. **fi** or **do**..**od** constructs. The latter executes the former repeatedly until the **break** statement is executed. Listing 2 shows two examples of the constructs. The first where **else** is taken only when there is no other enabled guards and another where an always enabled condition might never be executed.

Listing 2. Example of control flows in Promela.

```
if
:: (A == true) -> printf("A is true, B is unknown");
:: (B == true) -> printf("B is true, A is unknown");
:: else -> printf("A and B are false");
fi

do
:: (skip)->
      printf("If A is always true, then this may never printed.");
      break; /* breaks the do loop */
:: (A == true) ->
      printf("A is true");
      i = i + 1;
od
```

If the SPIN model checker performs an automatic verification of the above code, then it will visit every possible state until it aborts with the error: "max search depth too small". The reason is that, there is no deterministic set of values for i, thus the system state space can never be completely explored. It is crucial that all control flows have a valid end-state otherwise SPIN can not verify the model.

The SPIN model checker can verify models written in Promela. In 1986, Vardi and Wolper [3] published the foundation for SPIN, an automata-theoretic approach to automatic program verification. SPIN [4] can verify a model for correctness by generating a C program that performs an exhaustive verification of the system state space. During simulation and verification SPIN checks for the absence of deadlocks, livelocks, race conditions, unspecified receptions and unexecutable code.

The model checker can also be used to show the correctness of system invariants, find non-progress execution cycles and linear time temporal constraints, though we have not used any of those features for the model checking in this paper.

1. Related Work

Various possibilities for synchronous communication can be found in most network libraries, but we focus exclusively on network-enabled communication libraries that support Hoare's CSP algebra [5,6]. Several projects have investigated how to do CSP in a distributed environment. JCSP [7], Pony/occam-π [8] and C++CSP [9] provide network-enabled channels. Common to all three is that they use a specific naming for the channels, such that channels are reserved for one-to-one, one-to-any, network-enabled and so on. JCSP and C++CSP2 have the limitation that they can only do external choice (alt) on some channel types.

Pony enables transparent network support for occam-π. Schweigler and Sampson [8] write: "As long as the interface between components (i.e. processes) is clearly defined, the programmer should not need to distinguish whether the process on the other side of the interface is located on the same computer or on the other end of the globe". Unfortunately the pony implementation in occam-π is difficult to use as basis for a CSP library in languages like C++, Java or Python, as it relies heavily on the internal workings of occam-π. Pony/occam-π does

not currently have support for networked buffered channels. The communication overhead in Python is quite high, thus we are especially interested in fast one-to-one buffered networked channels, because they have the potential to hide the latency of the network. This would, for large parallel computations, make it possible to overlap computation with communication.

2. The Dynamic Channel

We present the basis for a dynamic channel type that combines multiple channel synchronisation mechanisms. The interface of the dynamic channel resembles a single channel type. When the channel is first created, it may be an any-to-any specialised for co-routines. The channel is then upgraded on request, depending on whether it participates in an alt and on the number of channel-ends connected. The next synchronisation level for the channel may be an optimised network-enabled one-to-one with no support for alt. Every upgrade stalls the communication on the channel momentarily while all active requests for a read or write are transformed to a higher synchronisation level. The upgrades continue, until the lowest common denominator (a network-enabled any-to-any with alt support) is reached.

This paper presents three models that are crucial parts in the dynamic channel design. These are: a local channel synchronisation model for shared memory, a distributed synchronisation model and the model for on-the-fly switching between synchronisation levels. We have excluded the following features to avoid state-explosion during automatic verification: mobility of channel ends, termination handling, buffered channels, skip / timeout guards and a discovery service for channel homes. Basically, we have simplified a larger model as much as possible and left out important parts, to focus on the synchronisation model handling the communication.

The different models are written in Promela to verify the design using the SPIN model checker. The verification phase is presented in section 3 where the three models are model-checked successfully. The full model-checked models are available at the PyCSP repository [10]. After the following overview, the models are described in detail:

- the local synchronisation model is built around the two-phase locking protocol. It provides a single CSP channel type supporting any-to-any communication with basic read / write and external choice (alt).
- the distributed synchronisation model is developed from the local model, providing the same set of constructs. The remote communication is similar to asynchronous sockets.
- the transition model enables the combination of a local (and faster) synchronisation model with more advanced distributed models. Channels are able to change synchronisation mechanisms, for example based on the location of channel ends, making it a dynamic channel.

For all models presented we do not handle operating system errors that cause threads to terminate or lose channel messages. We assume that all models are implemented on top of systems that provide reliable threads and message protocols.

2.1. Channel Synchronisation with Two-Phase Locking

The channel model presented here is similar to the PyCSP implementation (threads and processes) from 2009 [11] and will work as a verification of the method used in [11,12]. It is a single CSP channel type supporting any-to-any communication with basic read / write and external choice (alt).

In figure 1 we show an example of how the matching of channel operations comes about. Four processes are shown communicating on two channels using the presented design for negotiating read, write and external choice. Three requests have been posted to channel A

and two requests to channel B. During an external choice, a request is posted on multiple channels. Process 2 has posted its request to multiple channels and has been been matched. Process 1 is waiting for a successful match. Process 3 has been matched and is going to remove its request. Process 4 is waiting for a successful match. In the future, process 1 and process 4 are going to be matched. The matching is initiated by both, but only one process marks the match as successful.

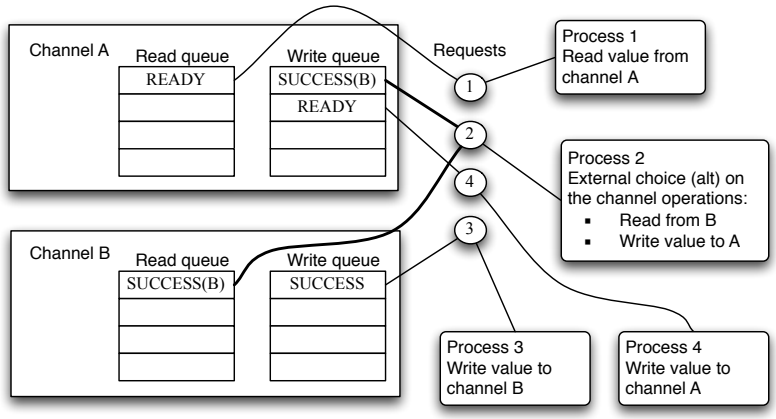

Figure 1. Example of four processes matching channel operations on two channels.

Listing 3. Simple model of a mutex lock with a condition variable. This is the minimum functionality, which can be expected from any multi-threading library.

```
typedef processtype {
        mtype state;
        bit lock;
        bit waitX;
};

processtype proc[THREADS];

inline acquire(lock_id) {
        atomic { (proc[lock_id].lock == 0); proc[lock_id].lock = 1; }
}
inline release(lock_id) {
        proc[lock_id].lock = 0;
}
inline wait(lock_id) {
        assert(proc[lock_id].lock == 1); /* lock must be acquired */
        atomic {
                release(lock_id);
                proc[lock_id].waitX = 0;     /* reset wait condition */
        }
        (proc[lock_id].waitX == 1); /* wait */
        acquire(lock_id);
}
inline notify(lock_id) {
        assert(proc[lock_id].lock == 1); /* lock must be acquired */
        proc[lock_id].waitX = 1;         /* wake up waiting process */
}
```

We use the two-phase locking protocol for channel synchronisation. When two processes are requesting to communicate on a channel, we accept the communication by first acquiring

the two locks, then checking the state of the two requests and if successful, updating and finally the two locks are released. This method requires many lock requests resulting in a large overhead, but it has the advantage that it never has to roll-back from trying to update a shared resource.

To perform the local synchronisation between threads, we implement the simple lock model shown in listing 3. This is straight-forward to model in Promela, as every statement in Promela must be executable and will block the executing thread until it becomes executable. The implemented lock model is restricted to single processes calling wait. If multiple processes called wait, then the second could erase a recent notify. For the models in the paper, we never have more than one waiting process on each lock.

Now that we can synchronise processes, the process state proc[id]. state can be protected on read and update. When blocked, we wait on a condition lock instead of wasting cycles using busy waiting, but the condition lock adds a little overhead. To avoid deadlocks, the process lock must be acquired before a process initiates a wait on a condition lock and before another process notifies the condition lock. The process calls wait in write (Listing 4) and is blocked until notified by offer (Listing 6). The offer function is called by the matching algorithm, which is initiated when a request is posted. To provide an overview, figure 2 shows a pseudo call graph of the model with all inline functions and the call relationship. A process can call read, write or alt to communicate on channels. These then posts the necessary requests to the involved channels and the matching algorithm calls offer for all matching pairs. Eventually a matching pair arrives at a success and the waiting process is notified.

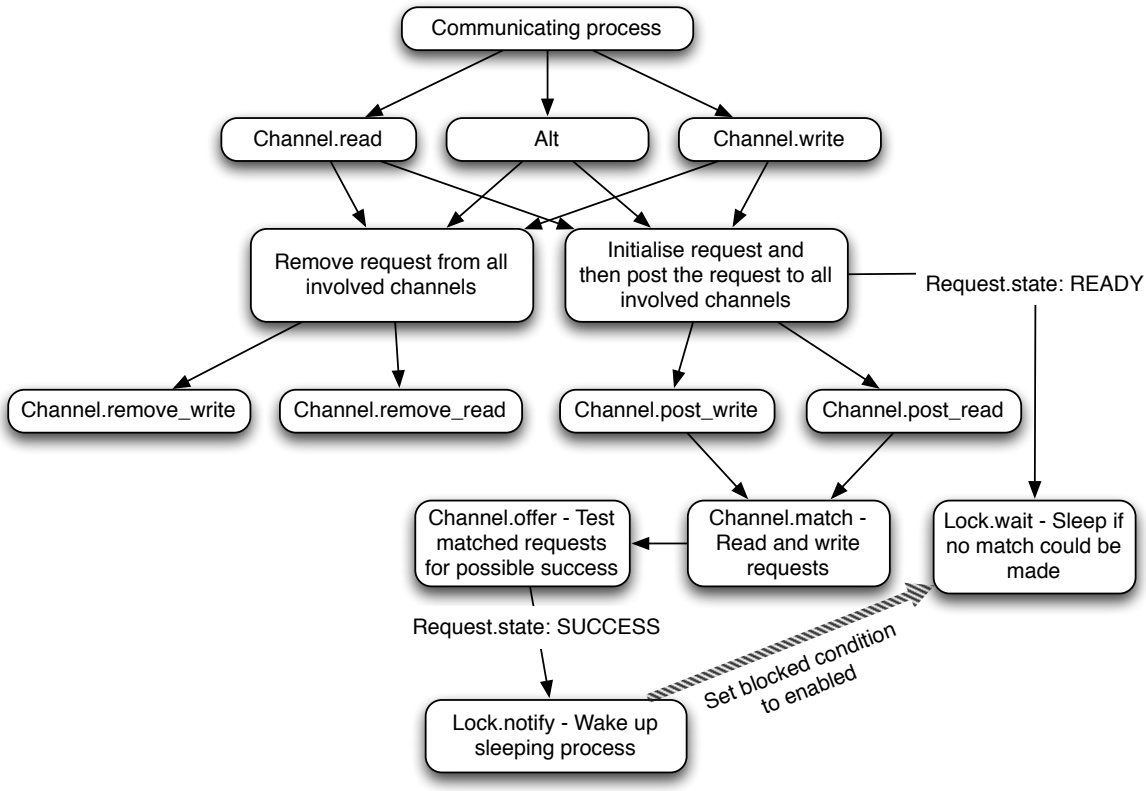

Figure 2. Pseudo call graph for the local channel synchronisation.

In write (Listing 4), a write request is posted to the write queue of the channel and again removed after a successful match with a write request. The corresponding functions read, post_read and remove_read are not shown since they are similar, except that remove_read returns the read value.

Listing 4. The write construct and the functions for posting and removing write requests. The process index _pid contains the Promela thread id.

```
inline write(ch_id, msg) {
  proc[_pid].state = READY;
  post_write(ch_id, msg);
  /* if no success, then wait for success */
  acquire(_pid);
  if
    :: (proc[_pid].state == READY) -> wait(_pid);
    :: else skip;
  fi;
  release(_pid);
  assert(proc[_pid].state == SUCCESS);
  remove_write(ch_id)
}
inline post_write(ch_id, msg_to_write) {
  /* acquire channel lock*/
  atomic { (ch[ch_id].lock == 0) -> ch[ch_id].lock = 1; }

  <add process id, msg_to_write to ch[ch_id].wqueue>
  match(ch_id);
  ch[ch_id].lock = 0; /* release channel lock */
}
inline remove_write(ch_id) {
  /* acquire channel lock*/
  atomic { (ch[ch_id].lock == 0) -> ch[ch_id].lock = 1; }

  <remove process id, msg from ch[ch_id].wqueue>
  ch[ch_id].lock = 0; /* release channel lock*/
}
```

When matching read and write requests on a channel we use the two-phase locking protocol where the locks of both involved processes are acquired before the system state is changed. To handle specific cases where multiple processes have posted multiple read and write requests, a global ordering of the locks (Roscoe's deadlock rule 7 [13]) must be used to make sure they are always acquired in the same order. In this local thread system we order the locks based on their memory address. This is both quick and ensures that the ordering never changes during execution. An alternative index for a distributed system would be to generate an index as a combination of the node address and the memory address.

Listing 5. Matching pairs of read and write requests for the two-phase locking.

```
inline match(ch_id) {
        w = 0; r = 0;
        do              /* Matching all reads to all writes */
        :: (r<ch[ch_id].rlen) ->
                w = 0;
                do
                :: (w<ch[ch_id].wlen) ->
                        offer(ch_id, r, w);
                        w = w+1;
                :: else break;
                od;
                r = r+1;
        :: else break;
        od;         }
```

The two-phase locking in offer (Listing 6) is executed for every possible pair of read and write requests found by match (Listing 5). The first phase acquires locks and the second phase releases locks. Between the two phases, updates can be made. Eventually when a matching is successful, three things are updated: the condition lock of both processes is notified, the message is transferred from the writer to the reader and proc[id]. state is updated.

One disadvantage of the two-phase locking is that we may have to acquire the locks of many read and write requests that are not in a ready state. The impact of this problem can easily be reduced by testing the state variable before acquiring the lock. Normally, this behaviour results in a race condition. However, the request can never change back to the ready state once it has been committed and remains posted on the channel. Because of this, the state can be tested before acquiring the lock, in order to find out whether time should be spent acquiring the lock. When the lock is acquired, the state must be checked again to ensure the request is still in the ready state. PyCSP [10] uses this approach in a similar offer method to reduce the number of acquired locks.

Listing 6. The offer function offering a possible successful match between two requests.

```
inline offer(ch_id, r, w) {
        r_pid = ch[ch_id].rqueue[r].id;
        w_pid = ch[ch_id].wqueue[w].id;
        if /* acquire locks using global ordering */
        :: (r_pid < w_pid) ->
                acquire(r_pid); acquire(w_pid);
        :: else skip ->
                acquire(w_pid); acquire(r_pid);
        fi;
        if /* Does the two processes match? */
        :: (proc[r_pid].state == READY && proc[w_pid].state == READY) ->
                proc[r_pid].state = SUCCESS;
                proc[w_pid].state = SUCCESS;

                /* Transfer message */
                ch[ch_id].rqueue[r].msg = ch[ch_id].wqueue[w].msg;
                ch[ch_id].wqueue[w].msg = NULL;
                proc[r_pid].result_ch = ch_id;
                proc[w_pid].result_ch = ch_id;

                notify(r_pid);
                notify(w_pid);
                 /* break match loop by updating w and r */
                w = LEN; r = LEN;
        :: else skip;
        fi;
        if /* release locks using reverse global ordering */
        :: (r_pid < w_pid) ->
                release(w_pid); release(r_pid);
        :: else skip ->
                release(r_pid); release(w_pid);
        fi;
}
```

The alt construct shown in listing 7 is basically the same as a read or write, except that the same process state is posted to multiple channels, thus ensuring that only one will be matched.

The alt construct should scale linearly with the number of guards. For the verification of the model we simplify alt to only accept two guards. If the model is model-checked success-

fully with two guards we expect an extended model to model-check successfully with more than two guards. Adding more guards to the alt construct in listing 7 is a very simple task, but it enlarges the system state-space and is unnecessary for the results presented in this paper.

Listing 7. The alt construct.

```
inline alt(ch_id1, op1, msg1, ch_id2, op2, msg2, result_chan, result) {
  proc[_pid].state = READY;
  result = NULL;
  if :: (op1 == READ) -> post_read(ch_id1);
     :: else             post_write(ch_id1, msg1);
  fi;
  if :: (op2 == READ) -> post_read(ch_id2);
     :: else             post_write(ch_id2, msg2);
  fi;
  acquire(_pid);        /* if no success, then wait for success */
  if :: (proc[_pid].state == READY) -> wait(_pid);
     :: else skip;
  fi;
  release(_pid);
  assert(proc[_pid].state == SUCCESS);
  if :: (op1 == READ) -> remove_read(ch_id1, result);
     :: else             remove_write(ch_id1);
  fi;
  if :: (op2 == READ) -> remove_read(ch_id2, result);
     :: else             remove_write(ch_id2);
  fi;
  result_chan = proc[_pid].result_ch;
}
```

2.2. Distributed Channel Synchronisation

The local channel synchronisation described in the previous section has a process waiting until a match has been made. The matching protocol performs a continuous two-phase locking for all pairs, thus the waiting process is constantly being tried even though it is passive. This method is not possible in a distributed model with no shared memory, instead an extra process is created to function as a remote lock, protecting updates of the posted channel requests. Similar to the local channel synchronisation, we must lock both processes in the offer function and retrieve the current process state from the process. Finally, when a match is found, both processes are notified and their process states are updated.

In figure 3, an overview of the distributed model is shown. The communicating process can call read, write or alt to communicate on channels. These then post the necessary requests to the involved channels through a Promela message channel. The channel home (channelThread) receives the request and initiates the matching algorithm to search for a successful offer amongst all matching pairs. During an offer, the channel home communicates with the lock processes (lockThread) to ensure that no other channel home conflicts. Finally, a matching pair arrives at a success and the lock process can notify the waiting process.

In listing 8 all Promela channels are created with a buffer size of 10 to model an asynchronous connection. We have chosen a buffer size of 10, as this is large enough to never get filled during verification in section 3. Every process communicating on a channel is required to have a lock process (Listing 9) associated, to handle the socket communication going in on proc_* **chan** types.

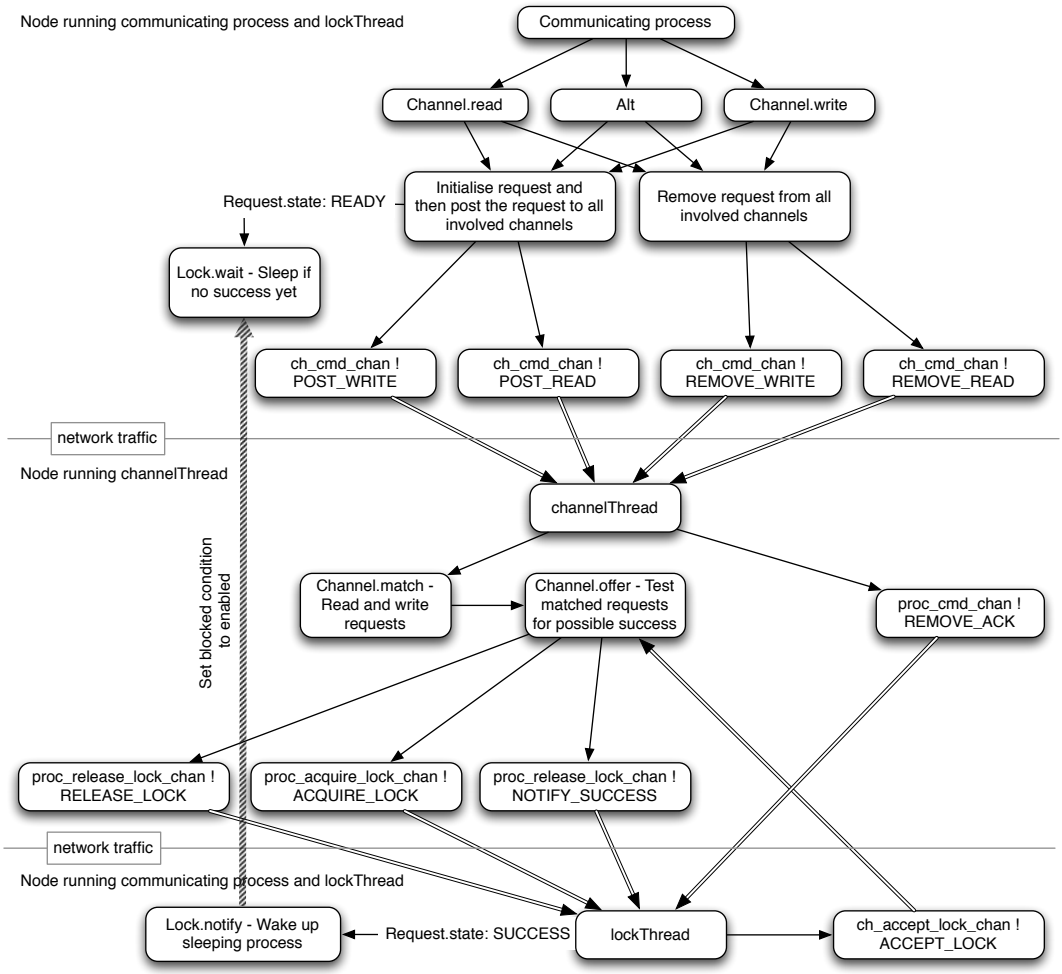

Figure 3. Pseudo call graph for the distributed channel synchronisation.

Listing 8. Modeling asynchronous sockets.

```
/* Direction: communicating process -> channelThread */
chan ch_cmd_chan[C]=[10] of {byte, byte, byte}; /*cmd, pid, msg*/
#define POST_WRITE 1
#define POST_READ 2
#define REMOVE_WRITE 3
#define REMOVE_READ 4

/* Direction: channelThread -> communicating process */
chan proc_cmd_chan[P]=[10] of {byte, byte, byte}; /*cmd, ch, msg*/
#define REMOVE_ACK 9

/* Direction: channelThread -> lockThread */
chan proc_acquire_lock_chan[P]=[10] of {byte}; /*ch*/

/* Direction: lockThread -> channelThread */
chan ch_accept_lock_chan[C]=[10] of {byte, byte}; /*pid, proc_state*/

/* Direction: channelThread -> lockThread */
chan proc_release_lock_chan[P]=[10] of {byte, byte, byte} /*cmd, ch, msg*/
#define RELEASE_LOCK 7
#define NOTIFY_SUCCESS 8
```

The lockThread in listing 9 handles the remote locks for reading and updating the process state from the channel home thread. The two functions remote_acquire and remote_release are called from the channel home process during the offer procedure. The lockThread and the communicating process use the mutex lock operations from listing 3 for synchronisation.

Listing 9. The lock process for a communicating process.

```
proctype lockThread(byte id) {
        byte ch_id, cmd, msg;
        byte ch_id2;
        bit locked;
        do
        :: proc_acquire_lock_chan[id]?ch_id ->
                ch_accept_lock_chan[ch_id]!id, proc[id].state;
                locked = 1;
                do
                :: proc_release_lock_chan[id]?cmd, ch_id2, msg; ->
                        if
                        :: cmd == RELEASE_LOCK ->
                                assert(ch_id == ch_id2);
                                break;
                        :: cmd == NOTIFY_SUCCESS ->
                                assert(ch_id == ch_id2);
                                acquire(id);  /* mutex lock op */
                                proc[id].state = SUCCESS;
                                proc[id].result_ch = ch_id2;
                                proc[id].result_msg = msg;
                                notify(id);   /* mutex lock op */
                                release(id);  /* mutex lock op */
                        fi;
                od;
                locked = 0;
        :: proc_cmd_chan[id]?cmd, ch_id, msg ->
                if
                :: cmd == REMOVE_ACK ->
                        proc[id].waiting_removes--;
                fi;
        :: timeout ->
                assert(locked == 0);
                assert(proc[id].waiting_removes == 0);
                break;
        od;
}
inline remote_acquire(ch_id, lock_pid, get_state) {
        proc_acquire_lock_chan[lock_pid]!ch_id;
        ch_accept_lock_chan[ch_id]?id, get_state;
        assert(lock_pid == id);
}
inline remote_release(ch_id, lock_pid) {
        proc_release_lock_chan[lock_pid]!RELEASE_LOCK, ch_id, NULL;
}
```

The offer function in listing 10 performs a distributed version of the function in listing 6. In this model we exchange the message from the write request to the read request, update the process state to SUCCESS, notifies the condition lock and release the lock process, all in one transmission to the Promela channel proc_release_lock_chan . We may still have to acquire the locks of many read and write requests that are not in ready state. Acquiring the locks are now more expensive than for the local channel model and it would happen more often, due to the latency of getting old requests removed. If an extra flag is added to a request the offer function can update the flag on success. If the flag is set, we know that the request has already been accepted and we avoid the extra remote lock operations. If the flag is not set, the request may still be old and not ready, as it might have been accepted by another process.

Listing 10. The offer function for distributed channel communication.

```
inline offer(ch_id, r, w) {
        r_pid = ch[ch_id].rqueue[r].id;
        w_pid = ch[ch_id].wqueue[w].id;
        if /* acquire locks using global ordering */
        :: (r_pid < w_pid) ->
                remote_acquire(ch_id, r_pid, r_state);
                remote_acquire(ch_id, w_pid, w_state);
        :: else skip ->
                remote_acquire(ch_id, w_pid, w_state);
                remote_acquire(ch_id, r_pid, r_state);
        fi;
        if /* Does the two processes match? */
        :: (r_state == READY && w_state == READY) ->
                proc_release_lock_chan[r_pid]!
                        NOTIFY_SUCCESS, ch_id, ch[ch_id].wqueue[w].msg;
                proc_release_lock_chan[w_pid]!
                        NOTIFY_SUCCESS, ch_id, NULL;
                w = LEN; r = LEN; /* break match loop */
        :: else skip;
        fi;
        if /* release locks using reverse global ordering */
        :: (r_pid < w_pid) ->
                remote_release(ch_id, w_pid);
                remote_release(ch_id, r_pid);
        :: else skip ->
                remote_release(ch_id, r_pid);
                remote_release(ch_id, w_pid);
        fi;
}
```

Every channel must have a channel home, where the read and write requests for communication are held and the offers are made. The channel home invokes the matching algorithm for every posted request, as the post_* functions did in the local channel model. In this model every channel home is a process (Listing 11). In another implementation there might only be one process per node maintaining multiple channel homes through a simple channel dictionary.

Listing 11. The channel home process.

```
proctype channelThread(byte ch_id) {
  DECLARE_LOCAL_CHANNEL_VARS
  do
  :: ch_cmd_chan[ch_id]?cmd, id, msg ->
     if
     :: cmd == POST_WRITE ->
       <add process id, msg to ch[ch_id].wqueue>
       match(ch_id);

     :: cmd == POST_READ ->
       <add process id, msg to ch[ch_id].rqueue>
       match(ch_id);

     :: cmd == REMOVE_WRITE ->
       <remove process id, msg from ch[ch_id].wqueue>
       proc_cmd_chan[id]!REMOVE_ACK, ch_id, NULL;

     :: cmd == REMOVE_READ ->
       <remove process id, msg from ch[ch_id].rqueue>
       proc_cmd_chan[id]!REMOVE_ACK, ch_id, NULL;

     fi;
  :: timeout -> /* controlled shutdown */
     /* read and write queues must be empty */
     assert(ch[ch_id].rlen == 0 && ch[ch_id].wlen == 0);
     break;
od;
}
```

The functions read, write and alt are for the distributed channel model identical to the local channel model. We can now transfer a message locally using the local channel model or between nodes using the distributed channel model.

2.3. Dynamic Synchronisation Layer

The following model will allow channels to change the synchronisation mechanism on-the-fly. This means that a local channel can be upgraded to become a distributed channel. Activation of the upgrade may be caused by a remote process requesting to connect to the local channel. The model presented in this section can not detect which synchronisation mechanism to use, it must be set explicitly. If channel-ends were part of the implementation, a channel could keep track of the location of all channel-ends and thus it would know what mechanism to use.

A feature of the dynamic synchronisation mechanism is that specialised channels can be used, such as a low-latency one-to-one channel resulting in improved communication time and lower latency. The specialised channels may not support constructs like external-choice (alt), but if an external-choice occurs the channel is upgraded. The upgrade procedure adds an overhead, but since channels are often used more than once this is an acceptable overhead.

Figure 4 shows an overview of the transition model. In the figure, the communicating process calls read or write to communicate on channels. These then call the functions enter, wait and leave functions. The enter function posts the request to the channel. The wait function ensures that the post is posted at the correct synchronisation level, otherwise it calls the transcend function. The leave function is called, when the request has been matched successfully. The model includes a thread that at any time activates a switch in synchronisation level and thus may force a call to the transcend function.

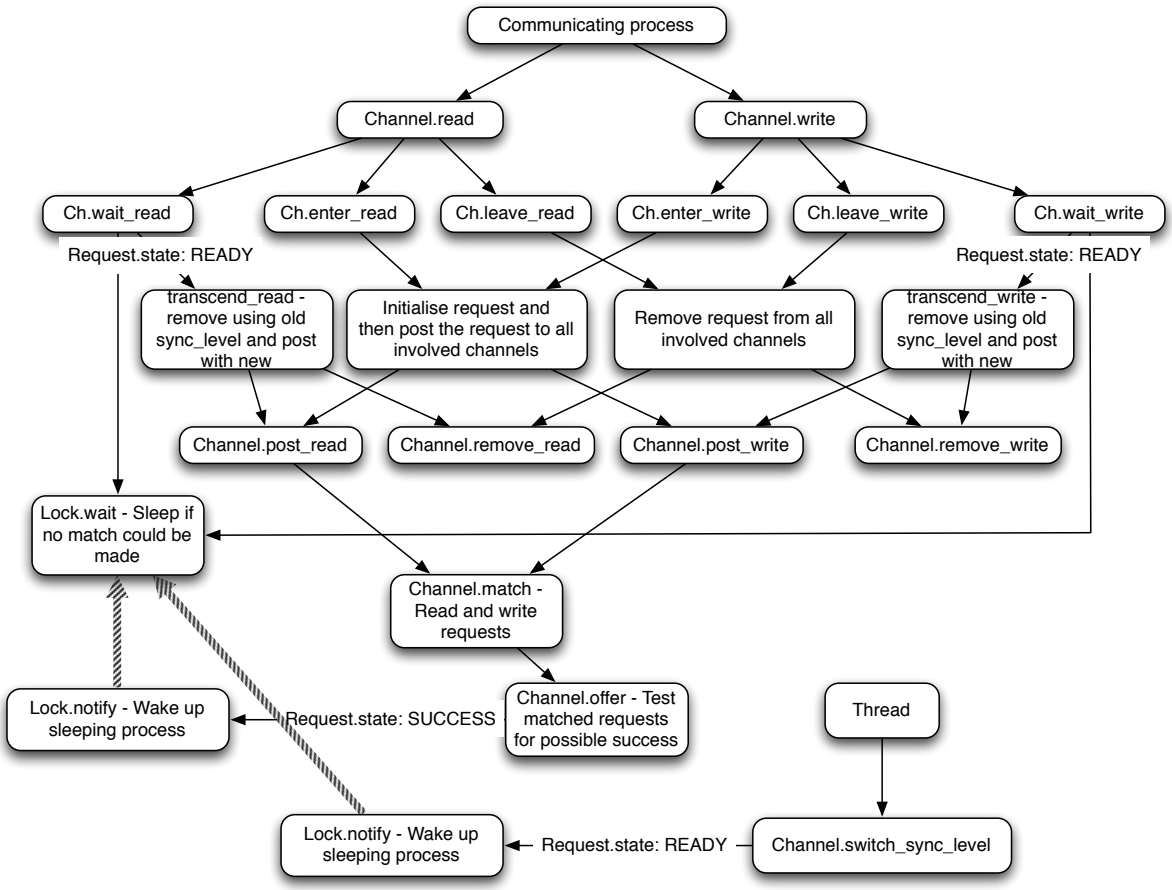

Figure 4. Pseudo call graph for the dynamic synchronisation layer.

To model the transition between two levels (layers) we set up two groups of channel request queues and a synchronisation level variable per channel. Every access to a channel variable includes the channel id and the new synchronisation level variable sync_level . Every communicating process is viewed as a single channel-end and is provided with a proc_sync_level . This way the communicating process will know the synchronisation level that it is currently at, even though the sync_level variable for the channel changes. The synchronisation level of a channel may change at any time using the switch_sync_level function in listing 12.

The match and offer functions from section 2.1 have been extended with a sync_level parameter used to access the channel container. The post_∗ functions update the proc_sync_level variable to the channel synchronisation level before posting a request, while the remove_∗ functions read the proc_sync_level variable and uses the methods of that level to remove the request. Other than that, the functions match, offer , post_∗ and remove_∗ are similar to the ones from the local channel model.

The switching of synchronisation level in listing 12 works by notifying all processes with a request for communication posted to the channel. The channel sync_level variable is changed before notifying processes. In listing 14 when a process either tries to enter wait or is awoken by the notification, it will check that the proc_sync_level variable of the posted request still matches the sync_level variable of the channel. If these do not match, we activate the transcend (Listing 13) function. During a transition, the proc_state variable is temporarily changed to SYNC, so that the request is not matched by another process between release and leave_read . The leave_read function calls remove_read which uses the proc_sync_level variable to remove the request and enter_read calls post_read which uses the updated channel sync_level variable.

Listing 12. Switching the synchronisation level of a channel.

```
inline switch_sync_level(ch_id, to_level) {
        byte SL;
        byte r,w,r_pid, w_pid;
        SL = ch[ch_id].sync_level;
        atomic {        (ch[ch_id].lvl[SL].lock == 0) ->
                        ch[ch_id].lvl[SL].lock = 1; } /* acquire */
        ch[ch_id].sync_level = to_level;

        /* Notify connected processes */
        r = 0;
        do
        :: (r<ch[ch_id].lvl[SL].rlen) ->
                r_pid = ch[ch_id].lvl[SL].rqueue[r];
                acquire(r_pid);
                if
                :: proc_state[r_pid] == READY ->
                        notify(r_pid); /*Notify process to transcend*/
                :: else -> skip;
                fi;
                release(r_pid);
                r = r+1;
        :: else break;
        od;
        w = 0;
        do
        :: (w<ch[ch_id].lvl[SL].wlen) ->
                w_pid = ch[ch_id].lvl[SL].wqueue[w];
                acquire(w_pid);
                if
                :: proc_state[w_pid] == READY ->
                        notify(w_pid); /*Notify process to transcend*/
                :: else -> skip;
                fi;
                release(w_pid);
                w = w+1;
        :: else break;
        od;
        ch[ch_id].lvl[SL].lock = 0; /* release */
}
```

Listing 13. The transition mechanism for upgrading posted requests.

```
inline transcend_read(ch_id) {
        proc_state[_pid] = SYNC;
        release(_pid);
        leave_read(ch_id);
        enter_read(ch_id);
        acquire(_pid);
}
```

In listing 14 the read function from the local channel model (Section 2.1) is split into an enter, wait and leave part. To upgrade blocking processes we use the transition mechanism in listing 13 which can only be used between an enter and a leave part. We require that all synchronisation levels must have an enter part, a wait / notify state and a leave part.

Listing 14. The read function is split into an enter, wait and leave part.

```
inline enter_read(ch_id) {
        proc_state[_pid] = READY;
        post_read(ch_id);
}
inline wait_read(ch_id) {
        /* if no success, then wait for success */
        acquire(_pid);
        do
        :: (proc_sync_level[_pid] == ch[ch_id].sync_level) &&
           (proc_state[_pid] == READY) ->
                wait(_pid);
        :: (proc_sync_level[_pid] != ch[ch_id].sync_level) &&
           (proc_state[_pid] == READY) ->
                transcend_read(ch_id);
        :: else break;
        od;
        release(_pid);
}
inline leave_read(ch_id){
        assert(proc_state[_pid] == SUCCESS ||
               proc_state[_pid] == SYNC);
        remove_read(ch_id);
}
inline read(ch_id) {
        enter_read(ch_id);
        wait_read(ch_id);
        leave_read(ch_id);
}
```

The three models presented can be used separately for new projects or they can be combined to the following: a CSP library for a high-level programming language where channel-ends are mobile and can be sent to remote locations. The channel is automatically upgraded, which means that the communicating processes can exist as co-routines, threads and nodes. Specialised channel implementations can be used without the awareness of the communicating processes. Any channel implementation working at a synchronisation level in the dynamic channel, must provide six functions to the dynamic synchronisation layer: enter_read, wait_read, leave_read, enter_write, wait_write and leave_write.

3. Verification Using SPIN

The commands in listing 15 verify the state-space system of a SPIN model written in Promela. The verification process checks for the absence of deadlocks, livelocks, race conditions, unspecified receptions, unexecutable code and user-specified assertions. One of these user-specified assertions checks that the message is correctly transferred for a channel communication. All verifications were run in a single thread on an Intel Xeon E5520 with 24 Gb DDR3 memory with ECC.

Listing 15. The commands for running an automatic verification of the models.

```
spin -a model.p
gcc -o pan -O2 -DVECTORSZ=4196 -DMEMLIM=24000 -DSAFETY \\
        -DCOLLAPSE -DMA=1112 pan.c
./pan
```

The local and the distributed channel models are verified for six process configurations and the transition model is verified for three process configurations. The results from running the SPIN model checker to verify models is listed in table 1. The automatic verification of the models found no errors. The "threads in model" column shows the threads needed for running the configuration in the specific model. The number of transitions in table 1 does not relate to how a real implementation of the model performs, but is the total amount of different transitions between states. If the number of transitions is high, then the model allows a large number of statements to happen in parallel. The SPIN model checker tries every transition possible, and if all transitions are legal the model is verified successfully for a process configuration. This means that for the verified configuration, the model has no deadlocks, no livelocks, no starvation, no race-conditions and do not fail with a wrong end-state.

The longest running verification which completed was the distributed model for the configuration in figure 5(f). This configuration completed after verifying the full state-space in 9 days. This means that adding an extra process to the model would multiply the total number of states to a level where we would not be able to complete a verification of the full state-space. The DiVinE model checker [14] is a parallel LTL model checker that should be able to handle larger models than SPIN, by performing a distributed verification. DiVinE has not been used with the models presented in this paper.

Table 1. The results from using the SPIN model checker to verify models.

Model	Configuration	Threads in model	Depth	Transitions
Local	Fig. 5(a)	2	91	1217
Local	Fig. 5(b)	2	163	10828
Local	Fig. 5(c)	3	227	149774
Local	Fig. 5(d)	4	261	2820315
Local	Fig. 5(e)	3	267	420946
Local	Fig. 5(f)	3	336	2056700
Distributed	Fig. 5(a)	5	151	90260
Distributed	Fig. 5(b)	6	245	28042640
Distributed	Fig. 5(c)	7	326	18901677
Distributed	Fig. 5(d)	9	446	1.1157292e+09
Distributed	Fig. 5(e)	8	406	6.771875e+08
Distributed	Fig. 5(f)	8	532	1.2102407e+10
Transition sync layer	Fig. 5(a)	3	162	43277
Transition sync layer	Fig. 5(c)	4	346	18567457
Transition sync layer	Fig. 5(d)	5	467	3.9206391e+09

The process configurations in figure 5 cover a wide variety of possible transitions for the local and distributed models. None of the configurations check a construct with more than two processes, but we expect the configurations to be correct for more than two processes. The synchronisation mechanisms are the same for a reading process and a writing process in the presented models. Based on this, we can expect that all the configurations in figure 5 can be mirrored and model-checked successfully. The local one-to-one communication is handled by the configuration in figure 5(a). Configurations in figure 5(c) and figure 5(d) cover the one-to-any and any-to-any cases, and we expect any-to-one to also be correct since it is a mirrored version of a one-to-any. The alt construct supports both input and output guards, thus figure 5(b) presents an obvious configuration to verify. In CSP networks this configuration does not make sense, but the verification of the configuration in figure 5(b) shows that two competing alts configured with the worst-case priority do not cause any live-locks. We must also model-check when alt communicates with reads or writes (Figure 5(e)).

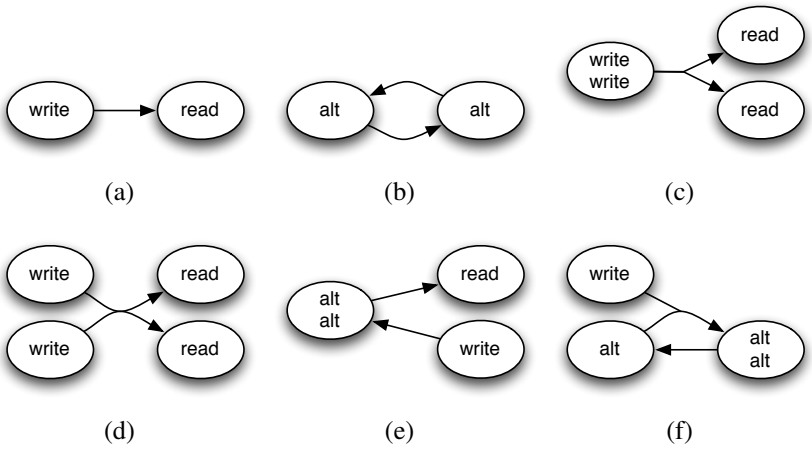

Figure 5. Process configurations used for verification.

Finally, the configuration in figure 5(f) verify when alts are communicating on one-to-any and any-to-one. These configurations cover most situations for up to two processes.

4. Conclusions

We have presented three building blocks for a dynamic channel capable of transforming the internal synchronisation mechanisms during execution. The change in synchronisation mechanism is a basic part of the channel and can come about at any time. In the worst case, the communicating processes will see a delay caused by having to repost a communication request to the channel.

Three models have been presented and model-checked: the shared memory channel synchronisation model, the distributed channel synchronisation model and the dynamic synchronisation layer. The SPIN model checker has been used to perform an automatic verification of these models separately. During the verification it was checked that the communicated messages were transferred correctly using assertions. All models were found to verify with no errors for a variety of configurations with communicating sequential processes. The full model of the dynamic channel has not been verified, since the large state-space may make it unsuited for exhaustive verification using a model checker.

With the results from this paper, we can also conclude that the synchronisation mechanism in the current PyCSP [11,12] can be model-checked succesfully by SPIN. The current PyCSP uses the two-phase locking approach with total ordering of locks, which has now been shown to work correctly for both the shared memory model and the distributed model.

4.1. Future Work

The equivalence between the dynamic channel presented in this paper and CSP channels, as defined in the CSP algebra, needs to be shown. Through equivalence, it can also be shown that networks of dynamic channels function correctly.

The models presented in this paper will be the basis for a new PyCSP channel, that can start out as a simple pipe and evolve into a distributed channel spanning multiple nodes. This channel will support mobility of channel ends, termination handling, buffering, scheduling of lightweight processes, skip and timeout guards and a discovery service for channel homes.

5. Acknowledgements

The authors would like to extend their gratitude for the rigorous review of this paper, including numerous constructive proposals from the reviewers.

References

[1] David Beazly. Understanding the Python GIL. `http://dabeaz.com/python/UnderstandingGIL.pdf`. Presented at PyCon 2010.

[2] Rune M. Friborg and Brian Vinter. Rapid Development of Scalable Scientific Software Using a Process Oriented Approach. *Journal of Computational Science*, page 11, March 2011.

[3] Moshe Y. Vardi and Pierre Wolper. An Automata-Theoretic Approach to Automatic Program Verification. *Proc. First IEEE Symp. on Logic in Computer Science*, pages 322–331, 1986.

[4] Gerard J. Holzman. The Model Checker Spin. *IEEE Trans. on Software Engineering*, pages 279–295, May 1997.

[5] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, pages 666–676, August 1978.

[6] C.A.R. Hoare. Communicating Sequential Processes. *Prentice-Hall*, 1985.

[7] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and Extending JCSP. In A.A.McEwan, S.Schneider, W.Ifill, and P.Welch, editors, *Communicating Process Architectures 2007*, Jul 2007.

[8] M. Schweigler and A. Sampson. p0ny - the occam-π Network Environment. *Communicating Process Architectures 2006*, pages 77–108, Jan 2006.

[9] Neil C. Brown. C++CSP Networked. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, pages 185–200, sep 2004.

[10] Pycsp distribution. `http://code.google.com/p/pycsp`.

[11] Rune M. Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In *Communicating Process Architectures 2009*, pages 277–292, 2009.

[12] Brian Vinter, John Markus Bjørndalen, and Rune M. Friborg. PyCSP Revisited. In *Communicating Process Architectures 2009*, pages 263–276, 2009.

[13] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International Series in Computer Science, 2005.

[14] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker. In *Parallel and Distributed Methods in Verification 2010*, pages 4–7, 2010.