COMMUNICATING SEQUENTIAL PROCESSES DOMAIN IN PTOLEMY II

by

Neil Smyth

Memorandum No. UCB/ERL M98/70

15 December 1998

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Masters Report

ABSTRACT

The Communicating Sequential Processes (CSP) domain in Ptolemy II models a system as a network of processes communicating with messages through unidirectional channels. The communication between processes is rendezvous based: both the reading and writing processes block until the other side is ready to communicate. This model of computation is nondeterministic and is also highly concurrent due to the nature of the model. This report gives an overview of the semantics of the model of computation of the CSP domain, the algorithms and software infrastructure used in implementing the domain, and some applications. In particular, applications for the CSP domain include resource management and high level system modeling early in the design cycle. Resource management is often required when modeling embedded systems, and to further support this, a notion of time has been added to the model of computation used in the domain.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Professor Edward Lee, for his support and guidance during my time in the Ptolemy group. I really enjoyed my interactions with him, and will always be grateful for showing me that there are many different ways to view the same problem.

Secondly, I would like to say thanks to all the members of the Ptolemy group, for the discussions and interactions that made my time in Berkeley such an enjoyable one.

This research is sponsored by the Defense Advanced Research Programs Agency (DARPA), the State of California MICRO program, and the following companies: cadence design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, Motorola, NEC, and Philips.

Finally, this report is dedicated to my parents and family, without whom none of this would have been possible.

1. **Introduction 1**
2. **Semantics of the Ptolemy II CSP Model 2**
   2.1. Rendezvous 2
   2.2. Conditional communication constructs 3
      2.2.1. CIF: 3
      2.2.2. CDO: 4
      2.2.3. Example using a CDO 4
   2.3. Deadlock 4
   2.4. Time 4
   2.5. Differences from original CSP model as proposed by Hoare 5
3. **Software infrastructure 6**
   3.1. Modeling in Ptolemy II 6
   3.2. CSP Domain 7
   3.3. Messages 9
4. **Using the CSP domain in Ptolemy II 10**
   4.1. Rendezvous 10
   4.2. Conditional communication constructs 10
   4.3. Time 13
5. **Model setup and control 14**
   5.1. Starting the model 14
   5.2. Detecting deadlocks: 15
   5.3. Terminating the model: 16
   5.4. Pausing/resuming the model 17
6. **Controlling communication between Threads 18**
   6.1. Brief introduction to threads in Java 18
      6.1.1. Approaches to locking used in the CSP domain 18
   6.2. Rendezvous algorithm 19
   6.3. Conditional Communication Algorithm 21
      6.3.1. Built on top of rendezvous: 21
      6.3.2. Choosing which branch succeeds 22
      6.3.3. Algorithm used by each branch: 22
   6.4. Modification of rendezvous algorithm: 24
7. **Demos and Examples 26**
   7.1. Dining Philosophers. 26
   7.2. M/M/1 27
   7.3. Pausing M/M/1 27
   7.4. Sieve of Eratosthenes 27
8. **Changes to the Topology during the Execution of a Model 28**
   8.1. How to write an actor that uses Topology Changes 28
   8.2. Sieve of Eratosthenes example 28
9. **Composing CSP with other domains 30**
   9.1. CSP inside another domain 30
      9.1.1. CSP within CSP 31
   9.2. Another domain inside CSP 31

10. Design decisions 33
10.1.Time: distributed relative time versus centralized absolute time 33
10.2.Choice of locks used and locking points 33
10.3.Making all the communication mechanisms symmetric 33
10.4.Conditional communication mechanisms upon rendezvous 33
10.5.Points in the model when changes to the topology are allowed 34
11. Conclusion and Future Work 35

1. Introduction

Ptolemy II is an environment that supports heterogeneous modeling and design of concurrent systems. Its focus is on embedded systems, particularly those that mix technologies. It offers a unified infrastructure for modeling systems using a number of models of computation. A *domain* executes a model of a system with the semantics of a particular model of computation. A model of a system may be designed using one domain, or it may choose to use several domains, hierarchically composed, to achieve greater accuracy or efficiency.

The Communicating Sequential Processes (CSP) domain in Ptolemy II models a system as a network of processes communicating with messages through unidirectional channels. If a process is ready to send a message, it blocks until the receiving process is ready to accept it. Similarly if a process is ready to accept a message, it blocks until the sending process is ready to send it. Thus the communication between processes is rendezvous based as both the reading and writing processes block until the other side is ready to communicate. This model of computation is non-deterministic as a process can be blocked waiting to send or receive on any number of channels. It is also highly concurrent due to the nature of the model.

The applications for the CSP domain include resource management and high level system modeling early in the design cycle. Resource management is often required when modeling embedded systems, and to further support this, a notion of time has been added to the model of computation used in the domain. This differentiates our CSP model from those more commonly encountered, which do not typically have any notion of time, although several versions of timed CSP have been proposed[6]. It might thus be more accurate to refer to the domain using our model of computation as the "Timed CSP" domain, but since the domain can be used with and without time, it is simply referred to as the CSP domain.

This report is written to be as self contained as possible, but invariably some details regarding how it builds upon the infrastructure in the Ptolemy II kernel and actor packages had to be omitted. For more details on these aspects, and on Ptolemy II in general, please refer to [13].

2. Semantics of the Ptolemy II CSP Model

The model of computation used in the CSP domain is based on the CSP model first proposed by Hoare[7] in 1978. In this model, a system is modeled as a network of processes communicating via messages along unidirectional channels. This in the only way processes can communicate, there is no shared state. The transfer of message between processes is via rendezvous, which means both the sending and receiving of a messages from a channel are blocking: i.e. the sending or receiving process stalls until the message is transferred. Some of the notation used here is borrowed from Andrews' book on concurrent programming [1], which refers to rendezvous-based message passing as synchronous message passing.

Process Networks (PN)[9] is a model of computation that has much in common with CSP. It also consists of a network of processes communicating via message passing along unidirectional channels. However, in PN, each channel has an unbounded first-in-first-out (FIFO) queue at the receiving end, so that the sending of messages along a channel is non-blocking. If there are no messages in the FIFO queue, then the receiving process stalls until a message is sent to the channel. The two models also differ in that PN is determinate whereas CSP is non-determinate due to the conditional communication constructs described below in section 2.2.

2.1 RENDEZVOUS

If a process is ready to send a message, it blocks until the receiving process is ready to accept it. Similarly if a process is ready to accept a message, it blocks until the sending process is ready to send it. Thus the communication between processes is rendezvous based as both the reading and writing processes block until the other side is ready to communicate. Figure 1 shows the case where one process is ready to send before the other process is ready to receive. The communication of information in this way can be viewed as a distributed assignment statement.

The sending process places some data in the message that it wants to send. The receiving process assigns the data in the message to a local variable. Of course, the receiving process may decide to ignore the contents of the message and only concern itself with the fact that a message arrived.

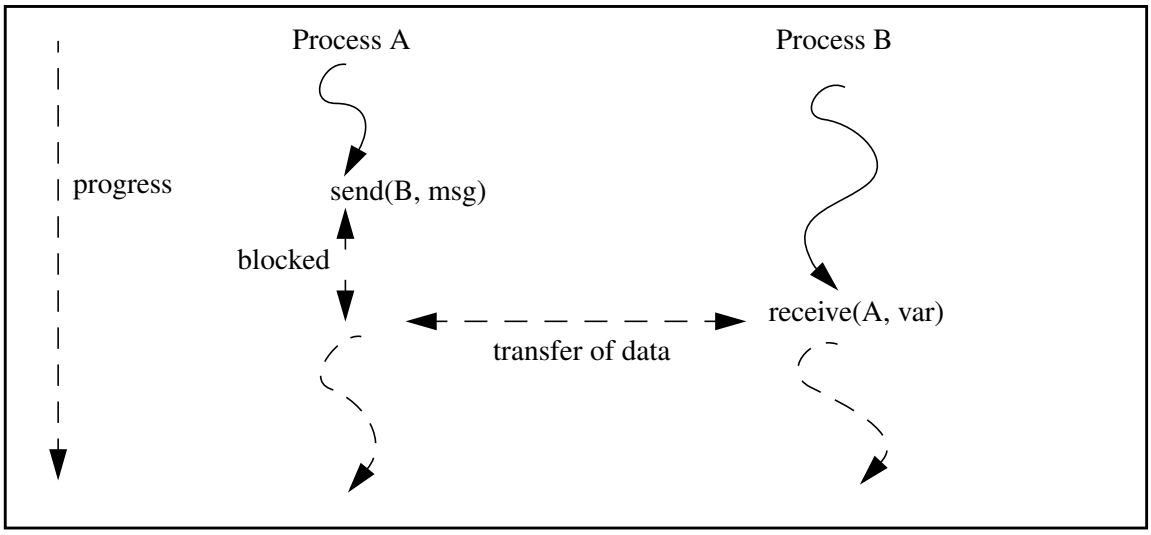


FIGURE 1. Illustrating how processes block waiting to rendezvous

2.2 CONDITIONAL COMMUNICATION CONSTRUCTS

A lot of the expressiveness in the CSP model comes from the being able to perform nondeterministic rendezvous. Nondeterministic rendezvous is based upon *guarded communication statements.*

A guarded communication statement has the form

```
guard; communication => statements;
```

The *guard* is only allowed to reference local variables, and its evaluation cannot changes the state of the process. For example it is not allowed to assign to variables, only reference them. The *communication* must be a simple send or receive, i.e. another conditional communication statement cannot be placed here. The *statements* part can contain any arbitrary sequence of statements, including more conditional communications.

If the guard is false, then the communication is not attempted and the statements are not executed. If the guard is true, then the communication is attempted, and if it succeeds, the following statements are executed. The guard may be omitted, in which case it is assumed to be true.

There are two conditional communication constructs built upon the guarded communication statements: **CIF** and **CDO**. These are analogous to the *if* and *while* statements in most programming languages. They should be read as “conditional if” and “conditional do”. Note that each guarded communication statement represents one *branch* of the CIF or CDO. The communication statement in each branch can be either a send or a receive, and they can be mixed freely.

2.2.1 CIF:

The form of a CIF is

```
CIF {
                G1;C1 => S1;
[]
                G2;C2 => S2;
[]
                ...
}
```

For each branch in the CIF, the guard (G1, G2,...) is evaluated. If it is true (or absent, which implies true), then the associated communication statement is enabled. If one or more branch is enabled, then the entire construct blocks until one of the communications succeeds. If more than one branch is enabled, the choice of which enabled branch succeeds with its communication is made nondeterministically. The successful communication is carried out, the associated statements are executed and the process continues. If all of the guards are false, then the process continues executing statements after the end of the CIF.

It is important to note that, although this construct is analogous to the common *if* programming construct, its behavior is very different. In particular all guards of the branches are evaluated concurrently, and the choice of which one succeeds does not depend on its position in the construct. The notation “[]” is used to hint at the parallelism in the evaluation of the guards. In a common *if* the branches are evaluated sequentially and the first branch that is evaluated to true is executed. The CIF construct also depends on the semantics of the communication between processes, and can thus stall the progress

of the thread if none of the enabled branches is able to rendezvous.

2.2.2 CDO:

The form of the CDO is

```
CDO {
        G1;C1 => S1;
[]
        G2;C2 => S2;
[]
        ...
}
```

The behavior of the CDO is similar to the CIF in that for each branch the guard is evaluated and the choice of which enabled communication to make is taken nondeterministically. However the CDO repeats the process of evaluating and executing the branches until *all* the guards return false. When this happens the process continues executing statements after the CDO construct.

2.2.3 Example using a CDO

An example use of a CDO is in a buffer process which can both accept and send messages, but has to be ready to do both at any stage. The code for this would look similar to that in figure 2. Note that in this case both guards can never be simultaneously false so this process will execute the CDO forever.

2.3 DEADLOCK

A deadlock situation is one in which none of the processes can make progress: they are all either blocked trying to rendezvous or they are delayed (see the next section). Thus two types of deadlock can be distinguished:

real deadlock - all active processes are blocked trying to communicate

time deadlock - all active processes are either blocked trying to communicate or are delayed, and at least one processes is delayed.

2.4 TIME

In the CSP domain, *time* is centralized. That is, all processes in a model share the same time,

```
CDO {
    (room in buffer?); receive(input, beginningOfBuffer) => update pointer to beginning of buffer;
[]
    (messages in buffer?); send(output, endOfBuffer) => update pointer to end of buffer;
}
```

FIGURE 2. Example of how a CDO might be used in a buffer

referred to as the *current model time*. Each process can only choose to *delay* itself relative for some period from the current model time, or a process can wait for time deadlock to occur at the current model time. Even though a process can be aware of the current model time, it should not choose to wait until the current model time reaches some value as the model time could change while it is waiting. It both cases, a process is said to be *delayed*.

When a process delays itself for some length of time from the current model time, it is suspended until time has sufficiently advanced, at which stage it wakes up and continues. If the process delays itself for zero time, this will have no effect and the process will continue executing. An example of the use of time in this manner can be seen below in section 7.2.

A process can also choose to delay its execution until the next occasion a time deadlock is reached. The process resumes at the same model time at which it delayed, and this is useful as a model can have several sequences of actions at the same model time. The next occasion time deadlock is reached, any processes delayed in this manner will continue, and time will not be advanced. An example of using time in this manner can be found in section 8.2.

Time may be *advanced* when all the processes are delayed or are blocked trying to rendezvous, and at least one process is delayed. If one or more processes are delaying until a time deadlock occurs, these processes are woken up and time is not advanced. Otherwise, the current model time is advanced just enough to wake up at least one process. Note that there is a semantic difference between a process delaying for zero time, which will have no effect, and a process delaying until the next occasion a time deadlock is reached.

Note also that time, as perceived by a single process, cannot change during its normal execution, only at rendezvous points or when the process delays. A process can be aware of the centralized time, but it cannot influence the current model time except by delaying itself. One of reasons behind using this model for time is given in 10.1. The choice for modeling time was in part influenced by Pamela[5], a run time library that is used to model parallel programs.

2.5 DIFFERENCES FROM ORIGINAL CSP MODEL AS PROPOSED BY HOARE

The model of computation used by the CSP domain differs from the original CSP[7] model in two ways. First, a notion of time has been added. The original proposal had no notion of time, although there have been several proposals for timed CSP[6]. Second, as mentioned in section 2.2, it is possible to use both send and receive in guarded communication statements. The original model only allowed receives to appear in these statements, though Hoare subsequently extended their scope to allow both communication primitives[8].

One final thing to note is that in much of the CSP literature, send is denoted using a "!", pronounced "bang", and receive is denoted using a "?", pronounced "query". This syntax was what was used in the original CSP paper[6] by Hoare. For example, the languages OCCAM[2] and LOTOS[3] both follow this syntax. In the CSP domain in Ptolemy II we use *send* and *get*, the choice of which is influenced by the desire to maintain uniformity of syntax across domains in Ptolemy II that use message passing. This supports the heterogeneity principle in Ptolemy II which enables the construction and interoperability of executable models that are built under a variety of models of computation. Similarly, the notation used in the CSP domain for conditional communication constructs differs from that commonly found in the CSP literature.

3. Software infrastructure

3.1 MODELING IN PTOLEMY II

In Ptolemy II an executable model consists of a top-level *CompositeActor* with an instance of *Director* and an instance of *Manager* associated with it. The manager provides overall control of the execution (starting, stopping, pausing). The director implements the semantics of the model of computation that governs the execution of *actors* contained by the CompositeActor.

The actors in the CompositeActor are connected to *Relations* via *Ports*. A relation connects one or more ports together. A particular collection of actors connected to each other through ports and relations is called a *topology*. The choice of the actors, the director controlling them and how they are connected defines what the model will do.

An actor under control of a director may be either an *AtomicActor*, which means it is indivisible, or it may be a CompositeActor, in which case it too can have its own director and contain a new set of actors. This is illustrated in figure 4(a).

Messages are passed between actors along relations. A relation has a *width*, greater than or equal to one. The width of a relation is the number of data *channels* represented by it. A port may have any number of relations connected to it, and the width of the port is defined to be the sum of the widths of the relations connected to it. If the port is an input port, it contains a set of receivers, one for each input channel. The receivers contained by a port are determined by the director controlling the model. A diagram illustrating how a message is transferred across a relation with one and two channels is shown in figure 4(b).

Obviously what has just been described is a very rough overview of the software infrastructure provided by Ptolemy II, though hopefully it is enough to allow the reader to understand the CSP models which are built on top of it. For a much more thorough description of Ptolemy II in general see [13].

3.2 CSP DOMAIN

In a CSP model, the director is an instance of *CSPDirector*. Since the model is controlled by a CSPDirector, all the receivers in the ports are *CSPReceivers*. The combination of the CSPDirector and CSPReceivers in the ports gives a model CSP semantics. The CSP domain associates each channel with exactly one receiver, located at the receiving end of the channel. Thus any process that sends or receives to any channel will rendezvous at a CSPReceiver. Figure 5 shows the static structure diagram of the five main classes in the CSP kernel, and a few of their associations. These are the classes that provide all the infrastructure needed for a CSP model.

Ptolemy II Syntax	**OCCAM syntax**
send	!
get	?
CIF	ALT
CDO	ALT wrapped in a while loop.

FIGURE 3. Comparison of syntaxes used in CSP domain and in OCCAM

CSPDirector: gives a model CSP semantics. It takes care of starting all the processes and controls/ responds to both real and time deadlocks. It also maintains and advances the model time when necessary.

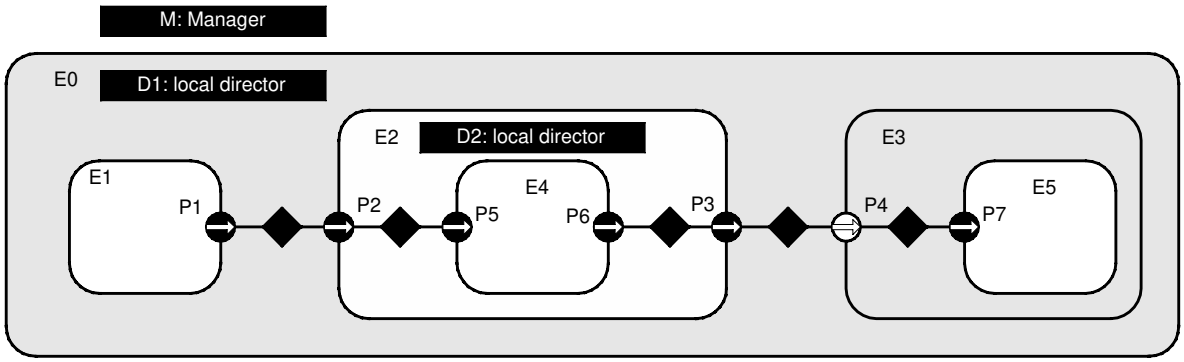
CSPReceiver: ensures that communication of messages between processes is via rendezvous.

CSPActor: adds the notion of time and the ability to perform conditional communication.

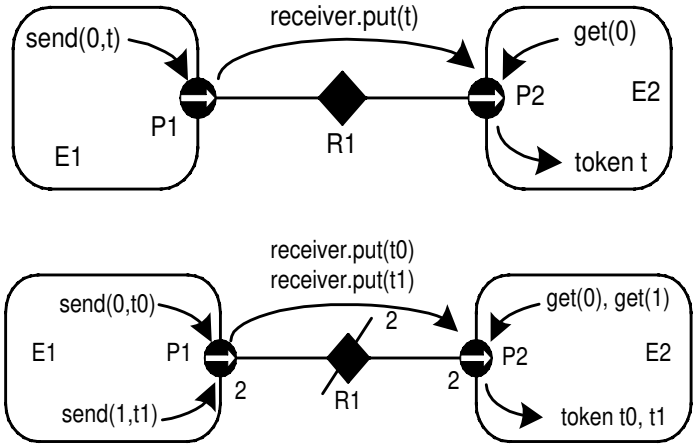
ConditionalReceive, ConditionalSend: used to construct the guarded communication statements necessary for the conditional communication constructs.

3.3 MESSAGES

All messages in Ptolemy II are represented by *Tokens*. The data carried in a message is defined by



(a)



(b)

FIGURE 4. (a) Example of a topology illustrating the control of a model and how the model may be hierarchically composed, (b) Detailed view of a relation with one and two channels in Ptolemy II.

the type of token used. The tokens available are shown in figure 6, though the user is free to develop new token classes. For more information on the token classes refer to [13]. For most models the tokens supplied should be sufficient.

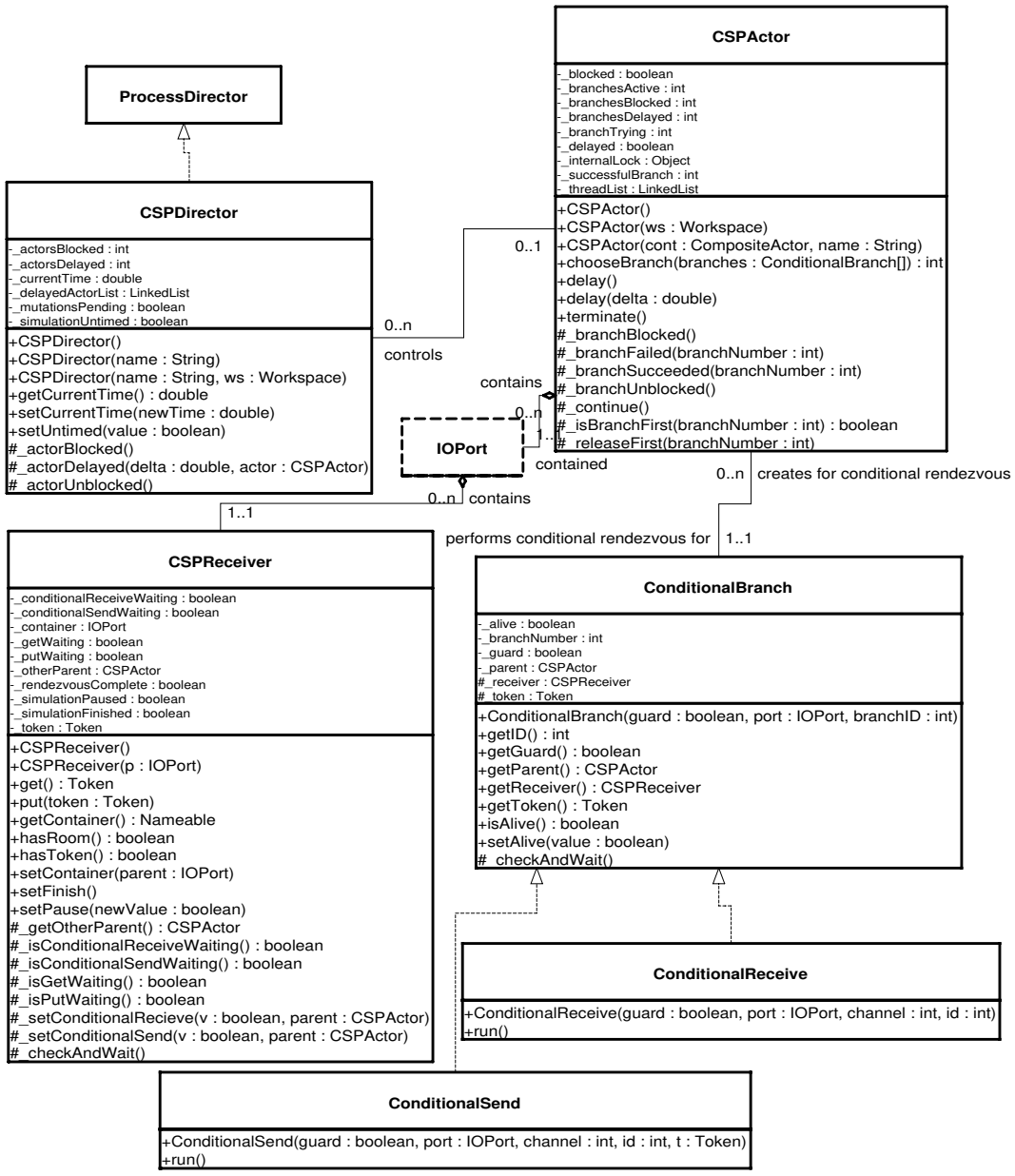


FIGURE 5. Static structure diagram for classes in the CSP kernel.

FIGURE 6. Tokens available in ptolemy.data package.

4. Using the CSP domain in Ptolemy II

For a model to have CSP semantics, it must have a CSPDirector controlling it. This ensures that the receivers in the ports are CSPReceivers, so all communication of messages between processes is via rendezvous. Note that each *actor* in the CompositeActor under the control of the CSPDirector represents a separate *process* in the model.

4.1 RENDEZVOUS

Since the ports contain CSPReceivers, the basic communication statements *send(channel, token)* and *get(channel)* will have rendezvous semantics. Thus the fact that a rendezvous is occurring on every communication is transparent to the actor code.

4.2 CONDITIONAL COMMUNICATION CONSTRUCTS

In order to use the conditional communication constructs, an actor must be derived from CSPActor. There are three steps involved:

1) Create a ConditionalReceive or ConditionalSend branch for each guarded communication statement, depending on the communication. Pass each branch a unique integer identifier, starting from zero, when creating it. The identifiers only need to be unique within the scope of that CDO or CIF.

2) Pass the branches to the chooseBranch() method in CSPActor. This method evaluates the guards, and decides which branch gets to rendezvous, performs the rendezvous and returns the identification number of the branch that succeeded. If all of the guards were false, -1 is returned.

3) Execute the statements for the guarded communication that succeeded.

```
boolean continueCDO = true;
while (continueCDO) {
        // step 1:
        ConditionalBranch[] branches = new ConditionalBranch[#branchesRequired];
        // Create a ConditionalReceive or a ConditionalSend for each branch
        // e.g. branches[0] = new ConditionalReceive( (guard), input, 0, 0);

        // step 2:
        int result = chooseBranch(branches);

        // step 3:
        if (result == 0) {
                // execute statements associated with first branch
        } else if (result == 1) {
                // execute statements associated with second branch.
        } else if ... // continue for each branch ID

        } else if (result == -1) {
                // all guards were false so exit CDO.
                continueCDO = false;
        } else {
```

FIGURE 7. Template for executing a CDO construct.

```
boolean guard = false;
boolean continueCDO = true;
ConditionalBranch[] branches = new ConditionalBranch[2];
while (continueCDO) {
        // step 1
        guard = (_size < depth);
        branches[0] = new ConditionalReceive(guard, input, 0, 0);
         guard = (_size > 0);
        branches[1] = new ConditionalSend(guard, output, 0, 1, _buffer[_readFrom]);

        // step 2
        int successfulBranch = chooseBranch(branches);

        // step 3
        if (successfulBranch == 0) {
                _size++;
                _buffer[_writeTo] = branches[0].getToken();
                 _writeTo = ++_writeTo % depth;
        } else if (successfulBranch == 1) {
                _size--;
                _readFrom = ++_readFrom % depth;
        } else if (successfulBranch == -1) {
           // all guards false so exit CDO
         // Note this cannot happen in this case
           continueCDO = false;
        } else {
           throw new TerminateProcessException(getName() + ": " +
                "branch id returned during execution of CDO.");
        }
}
```

FIGURE 8. Code used to implement the buffer process described in figure .

A sample template for executing a CDO is shown in figure 7. The code for the buffer described in figure 7 is shown in figure 8. In creating the ConditionalSend and ConditionalReceive branches, the first argument represents the guard. The second and third arguments represent the port and channel to send or receive the message on. The fourth argument is the identifier assigned to the branch. The choice of placing the guard in the constructor was made to keep the syntax of using guarded communication statements to the minimum, and to have the branch classes resemble the guarded communication statements they represent as closely as possible. This can give rise to the case where the Token specified in a ConditionalSend branch may not yet exist, but this has no effect as once the guard is false, the token in a ConditionalSend is never referenced.

The other option considered was to wrap the creation of each branch as follows:

```
if (guard) {
        // create branch and place in branches array
} else {
        // branches array entry for this branch is null
}
```

However this leads to longer actor code and what is happening is not as syntactically obvious.

The code for using a CIF is similar to the that in figure 7 except that the surrounding while loop is omitted and the case when the identifier returned is -1 does nothing. At some stage the steps involved in using a CIF or a CDO may be automated using a pre-parser, but for now the user must follow the approach described above.

It is worth pointing out that if most channels in a model are buffered, it may be worthwhile considering implementing the model in the PN domain which implicitly has an unbounded buffer on every channel.

4.3 TIME

If a process wishes to use time, the actor representing it must derive from CSPActor. As explained in section 2.4, each process in the CSP domain is able to delay itself, either for some period from the current model time or until the next occasion time deadlock is reached at the current model time. The two methods to call are delay(double) and waitForDeadlock(). Recall that if a process delays itself for zero time from the current time, the process will continue immediately. Thus delay(0.0) is not equivalent to waitForDeadlock()

If no processes are delayed, it is also possible to set the model time by calling the method setCurrentTime(newTime) on the director. However, this method can only be called when no processes are delayed, as the state of the model may be rendered meaningless if the model time is advanced to a time beyond the earliest delayed process. It is primarily for composing CSP with other domains, which is explained below in section 10.1.

As mentioned in section 2.4, as far as each process is concerned, time can only increase while it is blocked waiting to rendezvous or when delaying. A process can be aware of the current model time, but it should only ever affect the model time by delaying its execution, thus forcing time to advance. The method setCurrentTime(newTime) should never be called from a process.

By default every model in the CSP domain is timed. To use CSP without a notion of time, do not use the delay(double) method. The infrastructure supporting time does not affect the model execution if the delay(double) method is not used.

5. Model setup and control

The job of the CSPDirector in controlling the model is two fold. First, it must create and start a thread for each actor under its control. Each of these threads represents a process in our model. Second, it is responsible for detecting and responding to both real and time deadlocks. It can also pause and resume the model, and terminate all the processes when real deadlock is detected.

5.1 STARTING THE MODEL

The director creates a thread for each actor under its control in its initialize() method. It also invokes the initialize() method on each actor at this time. The director starts the threads in its prefire() method, and detects and responds to deadlocks in its fire() method. The thread for each actor is an instance of ProcessThread, which invokes the prefire(), fire() and postfire() methods for the actor until it finishes or is terminated. It then invokes the wrapup() method and the thread dies.

director.initialize() =>	create a thread for each actor update count of active processes with the director call initialize() on each actor	
director.prefire() =>	start the process threads =>	calls actor.prefire() calls actor.fire() calls actor.postfire() repeat.
director.fire() =>	handle deadlocks until a real deadlock occurs.	
director.postfire() =>	return a boolean indicating if the execution of the model should continue for another iteration	
director.wrapup() =>	terminate all the processes =>	calls actor.wrapup() decrease the count of active processes with the director

FIGURE 9. Sequence of steps involved in setting up and controlling the model.

Figure 10 shows the code executed by the ProcessThread class. Note that it makes no assumption about the actor it is executing, so it can execute any domain-polymorphic actor as well as CSP domain-specific actors. In fact any other domain actor that does not rely on the specifics of its parent domain can be executed in the CSP domain by the ProcessThread.

5.2 DETECTING DEADLOCKS:

For deadlock detection, the director maintains three counts:

- the number of *active* processes which are threads that have started but have not yet finished
- the number of *blocked* processes which is the number of processes that are blocked waiting to rendezvous, and

•the number of *delayed* processes which is the number of processes waiting for time to advance plus the number of processes waiting for time deadlock to occur at the current model time.

When the number of blocked processes equals the number of active processes, then real deadlock has occurred and the fire method of the director returns. When the number of blocked plus the number of delayed processes equals the number of active processes, and at least one process is delayed, then time deadlock has occurred. If at least one process is delayed waiting for time deadlock to occur at the current model time, then the director wakes up all such process and does not advance time. Otherwise the director looks at its list of processes waiting for time to advance, chooses the earliest one and advances time sufficiently to wake it up. It also wakes up any other processes due to be woken up at the new time. The director checks for deadlock each occasion a process blocks, delays or dies.

For the director to work correctly, these three counts need to be accurate at all stages of the model execution, so when they are updated becomes important. Keeping the active count accurate is relatively simple, the director increase it when it starts the thread, and decreases it when the thread dies. Likewise the count of delayed processes is straightforward: when a process delays, it increases the count of delayed processes, and the director keeps track of when to wake it up. The count is decreased when a delayed process resumes.

```
public void run() {
        try {
                boolean iterate = true;
                while (iterate) {
                        // container is checked for null to detect the termination
                        // of the actor.
                        iterate = false;
                        if ((Entity)_actor).getContainer() != null && _actor.prefire()) {
                                _actor.fire();
                                iterate = _actor.postfire();
                        }
                }
        } catch (TerminateProcessException t) {
                // Process was terminated early
        } catch (IllegalActionException e) {
                _manager.fireExecutionError(e);
        } finally {
                try {
                        _actor.wrapup();
                } catch (IllegalActionExeption e) {
                        _manager.fireExecutionError(e);
                }
                _director.decreaseActiveCount();
        }
}
```

FIGURE 10. Code executed by ProcessThread.run()

However, due to the conditional communication constructs, keeping the blocked count accurate requires a little more effort. For a basic send or receive, a process is registered as being blocked when it arrives at the rendezvous point before the matching communication. The blocked count is then decreased by one when the corresponding communication arrives. However what happens when an actor is carrying out a conditional communication construct? In this case the process keeps track of all of the branches for which the guards were true, and when all of those are blocked trying to rendezvous, it registers the process as being blocked. When one of the branches succeeds with a rendezvous, the process is registered as being unblocked.

5.3 TERMINATING THE MODEL:

A process can finish in one of two ways: either by returning false in its prefire() or postfire() methods, in which case it is said to have finished *normally*, or if it is terminated *early* by a TerminateProcessException being thrown. For example, if a source process is intended to send ten tokens and then finish, it would exit its fire() method after sending the tenth token, and return false in its postfire() method. This causes the ProcessThread, see figure 10, representing the process, to exit the while loop and execute the finally clause. The finally clause calls wrapup() on the actor it represents, decreases the count of active processes in the director, and the thread representing the process dies.

A TerminateProcessException is thrown whenever a process tries to communicate via a channel whose receiver has its *finished* flag set to true. When a TerminateProcessException is caught in ProcessThread, the finally clause is also executed and the thread representing the process dies.

To terminate the model, the director sets the *finished* flag in each receiver. The next occasion a process tries to send to or receive from the channel associated with that receiver, a TerminateProcessException is thrown. This mechanism can also be used in a selective fashion to terminate early any processes that communicate via a particular channel. When the director controlling the execution of the model detects real deadlock, it returns from its fire() method. In the absence of hierarchy, this causes the wrapup() method of the director to be invoked. It is the wrapup() method of the director that sets the finished flag in each receiver. Note that the TerminateProcessException is a runtime exception so it does not need to be declared as being thrown.

There is also the option of abruptly terminating all the processes in the model by calling terminate() on the director. This method differs from the approach described in the previous paragraph in that it stops all the threads immediately and does not give them a chance to update the model state. After calling this method, the state of the model is unknown and so the model should be recreated after calling this method. This method is only intended for situations when the execution of the model has obviously gone wrong, and for it to finish normally would either take too long or could not happen. It should rarely be called.

5.4 PAUSING/RESUMING THE MODEL

Pausing and resuming a model does not affect the outcome of a particular execution of the model, only the rate of progress. The execution of a model can be paused at any stage by calling the pause() method on the director. This method is blocking, and will only return when the model execution has been successfully paused. To pause the execution of a model, the director sets a *paused* flag in every receiver, and the next occasion a process tries to send to or receive from the channel associated with that receiver, it is paused. The whole model is paused when all the active processes are delayed, paused or blocked. To resume the model, the resume() method can similarly be called on the director This

method resets the paused flag in every receiver and wakes up every process waiting on a receiver lock. If a process was paused, it sees that it is no longer paused and continues. The ability to pause and resume the execution of a model is intended primarily for user interface control.

6. Controlling communication between Threads

6.1 BRIEF INTRODUCTION TO THREADS IN JAVA

The CSP domain, like the rest of Ptolemy II, is written entirely in Java and takes advantage of the features built into the language. In particular, the CSP domain depends heavily on *threads* and on *monitors* for controlling the interaction between threads. In any multi-threaded environment, care has to be taken to ensure that the threads do not interact in unintended ways, and that the model does not deadlock. Note deadlock in this sense is a bug in the *modeling environment*, which is different from the deadlock talked about before which may or may not be a bug in the *model* being executed.

A monitor is a mechanism for ensuring mutual exclusion between threads. In particular if a thread has a particular monitor, acquired in order to execute some code, then no other thread can simultaneously have that monitor. If another thread tries to acquire that monitor, it stalls until the monitor becomes available. A monitor is also called a *lock*, and one is associated with every object in Java.

Code that is associated with a lock is defined by the *synchronized* keyword. This keyword can either be in the signature of a method, in which case the entire method body is associated with that lock, or it can be used in the body of a method using the syntax:

```
synchronized(object) {
        // synchronized code goes here
}
```

This causes the code inside the brackets to be associated with the lock belonging to the specified object. In either case, when a thread tries to execute code controlled by a lock, it must either acquire the lock or stall until the lock becomes available. If a thread stalls when it already has some locks, those locks are not released, so any other threads waiting on those locks cannot proceed. This can lead to deadlock when all threads are stalled waiting to acquire some lock they need.

A thread can voluntarily relinquish a lock when stalling by calling *object.wait()* where *object* is the object to relinquish and wait on.This causes the lock to become available to other threads. A thread can also wake up any threads waiting on a lock associated with an object by calling notifyAll() on the object. Note that to issue a notifyAll() on an object it is necessary to own the lock associated with that object first. By careful use of these methods it is possible to ensure that threads only interact in intended ways and that deadlock does not occur.

6.1.1 Approaches to locking used in the CSP domain

One of the key coding patterns followed is to wrap each wait() call in a while loop that checks some flag. Only when the flag is set to false can the thread proceed beyond that point. Thus the code will often look like

```
synchronized(object) {
        ...
        while(flag) {
                object.wait();
        }
        ...
}
```

The advantage to this is that it is not necessary to worry about what other thread issued the notifyAll() on the lock; the thread can only continue when the notifyAll() is issued *and* the flag has been set to false.

Another approach used is to keep the number of locks acquired by a thread as few as possible, preferably never more than one at a time. If several threads share the same locks, and they must acquire more than one lock at some stage, then the locks should always be acquired in the same order. To see how this prevent deadlocks, consider two threads, *thread1* and *thread2*, that are using two locks A and B. If *thread1* obtains A first, then B, and *thread2* obtains B first then A, then a situation could arise whereby *thread1* owns lock A and is waiting on B, and *thread2* owns lock B and is waiting on A. Neither thread can proceed and so deadlock has occurred. This would be prevented if both threads obtained lock A first, then lock B. This approach is sufficient, but not necessary to prevent deadlocks, as other approaches may also prevent deadlocks without imposing this constraint on the program[10].

Finally, deadlock often occurs even when a thread, which already has some lock, tries to acquire another lock only to issue a notifyAll() on it. To avoid this situation, it is easiest if the notifyAll() is issued from a *new thread* which has no locks that could be held if it stalls. This is often used in the CSP domain to wake up any threads waiting on receivers, for example after a pause or when terminating the model. The class NotifyThread, in the ptolemy.actor.process package, is used for this purpose. This class takes a list of objects in a linked list, or a single object, and issues a notifyAll() on each of the objects from within a new thread.

The CSP domain kernel makes extensive use of the above patterns and conventions to ensure the modeling engine is deadlock free. However for a much more thorough introduction to concurrent programming Java, a very good starting point is [10].

6.2 RENDEZVOUS ALGORITHM

In CSP, the *locking point* for all communication between processes is the *receiver.* Any occasion a process wishes to send or receive, it must first acquire the lock for the receiver associated with the channel it is communicating over. Two key facts to keep in mind when reading the following algorithms are that each channel has exactly one receiver associated with it and that at most one process can be trying to send to (or receive from) a channel at any stage. The constraint that each channel can have at most one process trying to send to (or receive from) a channel at any stage is not currently enforced, but an exception will be thrown if such a model is not constructed.

The rendezvous algorithm is *entirely symmetric* for the put() and the get(), except for the direction the token is transferred. This helps reduce the deadlock situations that could arise and also makes the interaction between processes more understandable and easier to explain. The algorithm controlling how a get() proceeds is shown in figure 11. The algorithm for a put() is exactly the same except that put and get are swapped everywhere. Thus it suffices to explain what happens when a get() arrives at a receiver i.e. when a process tries to receive from the channel associated with the receiver.

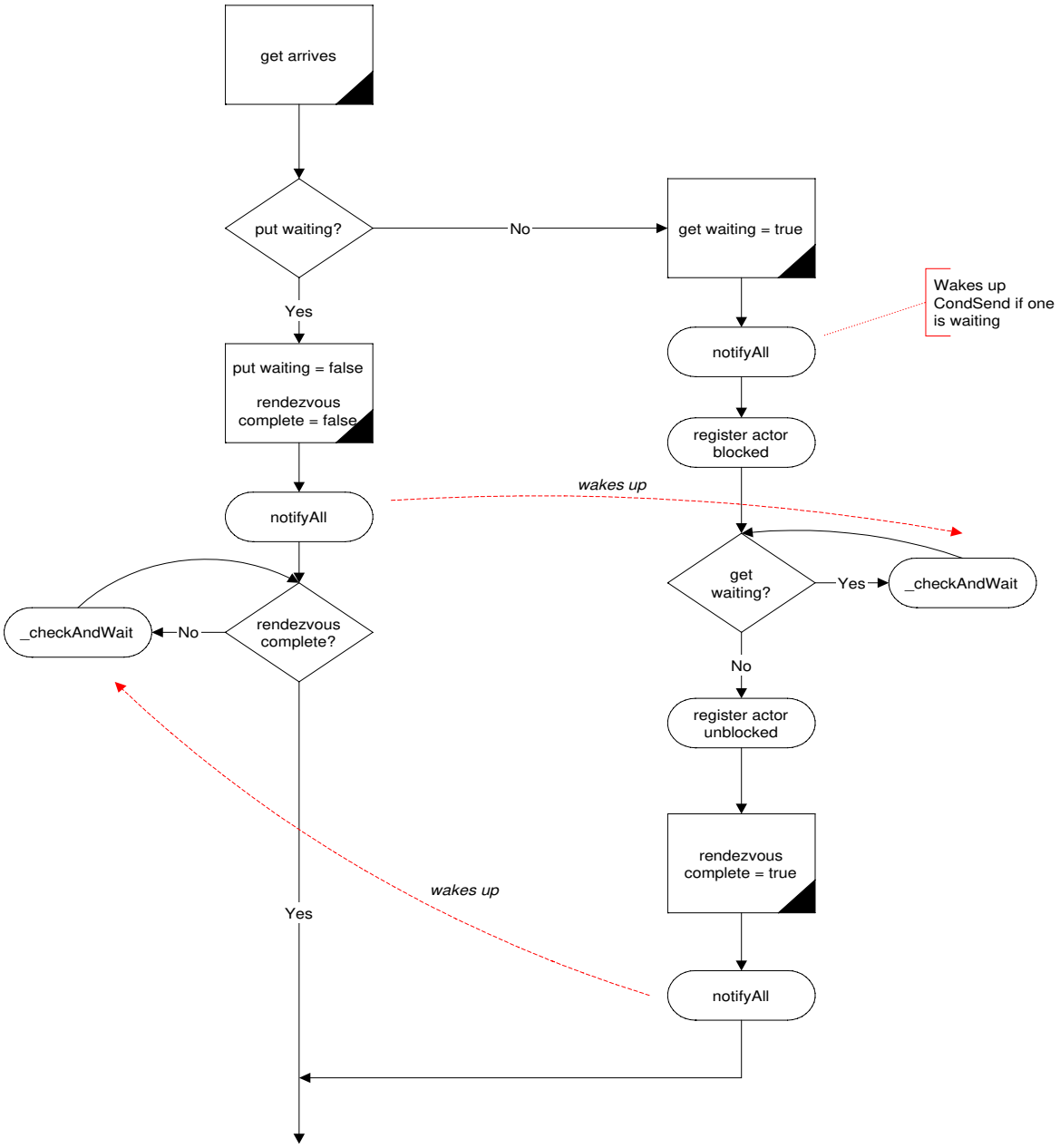


FIGURE 11. Rendezvous algorithm

When a get() arrives at a receiver, a put() is either already waiting to rendezvous or it isn't. Both the get() and put() methods are entirely synchronized on the receiver so they cannot happen simultaneously (only one thread can possess a lock at any given time). Without loss of generality assume a get() arrives before a put(). The rendezvous mechanism is basically three steps: a get() arrives, a put() arrives, the rendezvous completes.

(1) When the get arrives it sees that it is first and sets a flag saying a get is waiting. It then waits on the receiver lock while the flag is still true, *(2)* When a put arrives, it sets the *getWaiting* flag to false, wakes up any threads waiting on the receiver (including the get), sets the *rendezvousComplete* flag to false and then waits on the receiver while the *rendezvousComplete* flag is false, *(3)* The thread executing the get wakes up, sees that a put has arrived, sets the *rendezvousComplete* flag to true, wakes up any threads waiting on the receiver and returns thus releasing the lock. The thread executing the put then wakes up, acquires the receiver lock, sees that the rendezvous is complete and returns.

Following the rendezvous, the state of the receiver is exactly the same as before the rendezvous arrived, and it is ready to mediate another rendezvous. It is worth noting that the final step, of making sure the second communication to arrive does not return until the rendezvous is complete, is necessary to ensure that the correct token gets transferred. Consider the case again when a get arrives first, except now the put returns immediately if a get is already waiting. A put arrives, places a token in the receiver, sets the get waiting flag to false and returns. Now suppose another put arrives before the get wakes up, which will happen if the thread the put is in wins the race to obtain the lock on the receiver. Then the second put places a new token in the receiver and sets the put waiting flag to true. Then the get wakes up, and returns with the wrong token! This is known as a *race condition*, which will lead to unintended behavior in the model.

6.3 CONDITIONAL COMMUNICATION ALGORITHM

There are two steps involved in executing a CIF or a CDO: first deciding which enabled branch succeeds, then carrying out the rendezvous.

6.3.1 Built on top of rendezvous:

When a conditional construct has more than one enabled branch (guard is true or absent), a new thread is spawned for each enabled branch. The job of the chooseBranch() method is to control these threads and to determine which branch should be allowed to successfully rendezvous. These threads and the mechanism controlling them are entirely separate from the rendezvous mechanism described in section 6.2, with the exception of one special case, which is described in section 6.4. Thus the conditional mechanism can be viewed as being built on top of basic rendezvous: conditional communication knows about and needs basic rendezvous, but the opposite is not true. Again this is a design decision which leads to making the interaction between threads easier to understand and is less prone to deadlock as there are fewer interaction possibilities to consider.

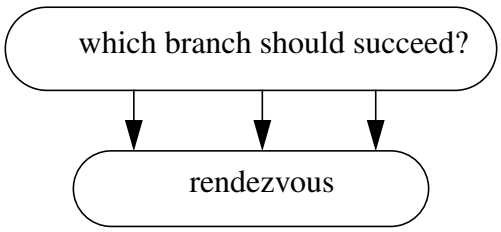


FIGURE 12. Conceptual view of how conditional communication is built on top of rendezvous.

6.3.2 Choosing which branch succeeds

The manner in which the choice of which branch can rendezvous is worth explaining. The chooseBranch() method in CSPActor takes an array of branches as an argument. If all of the guards are false, it returns -1, which indicates that all the branches failed. If exactly one of the guards is true, it performs the rendezvous directly and returns the identification number of the successful branch. The interesting case is when more than one guard is true. In this case, it creates and starts a new thread for each branch whose guard is true. It then waits, on an internal lock, for one branch to succeed. At that point it gets woken up, sets a finished flag in the remaining branches and waits for them to fail. When all the threads representing the branches are finished, it returns the identification number of the successful branch. This approach is designed to ensure that exactly one of the branches created successfully performs a rendezvous.

6.3.3 Algorithm used by each branch:

Similar to the approach followed for rendezvous, the algorithm by which a thread representing a branch determines whether or not it can proceed is entirely *symmetrical* for a ConditionalSend and a ConditionalReceive. The algorithm followed by a ConditionalReceive is shown figure 13. Again the locking point is the receiver, and all code concerned with the communication is synchronized on the receiver. The receiver is also where all necessary flags are stored.

Consider three cases.

(1) a conditionalReceive arrives and a put is waiting.

In this case, the branch checks if it is the first branch to be ready to rendezvous, and if so, it is goes ahead and executes a get. If it is not the first, it waits on the receiver. When it wakes up, it checks if it is still alive. If it is not, it registers that it has failed and dies. If it is still alive, it starts again by trying to be the first branch to rendezvous. Note that a put cannot disappear.

(2) a conditionalReceive arrives and a conditionalSend is waiting

When both sides are conditional branches, it is up to the branch that arrives second to check whether the rendezvous can proceed. If both branches are the first to try to rendezvous, the conditionalReceive executes a get(), notifies its parent that it succeeded, issues a notifyAll() on the receiver and dies. If not, it checks whether it has been terminated by chooseBranch(). If it has, it registers with chooseBranch() that it has failed and dies. If it has not, it returns to the start of the algorithm and tries again. This is because a ConditionalSend could disappear. Note that the parent of the first branch to arrive at the receiver needs to be stored for the purpose of checking if both branches are the first to arrive.

This part of the algorithm is somewhat subtle. When the second conditional branch arrives at the rendezvous point it checks that *both* sides are the first to try to rendezvous for their respective processes. If so, then the conditionalReceive executes a get(), so that the conditionalSend is never aware that a conditionalReceive arrived: it only sees the get().

(3) a conditionalReceive arrives first.

It sets a flag in the receiver that it is waiting, then waits on the receiver. When it wakes up, it checks if it has been killed by chooseBranch. If it has it registers with chooseBranch that it has failed and dies. Otherwise it checks if a put is waiting. It only needs to check if a put is waiting because if a conditionalSend arrived, it would have behaved as in case (2) above. If a put is waiting, the branch checks if it is the first branch to be ready to rendezvous, and if so it is goes ahead and executes a get. If

it is not the first, it waits on the receiver and tries again.

6.4 MODIFICATION OF RENDEZVOUS ALGORITHM:

Consider the case when a conditional send arrives before a get. If all the branches in the conditional communication which the conditional send is a part of are blocked, then the process will register itself as blocked with the director. Then the get comes along, and even though a conditional send is waiting, it too would register itself as blocked. This leads to one too many processes being registered as blocked, which could lead to premature deadlock detection.

To avoid this, it is necessary to modify the algorithm used for rendezvous slightly. The change to

FIGURE 13. Algorithm used to determine if a conditional rendezvous branch succeeds or fails

the algorithm is shown in the dashed ellipse in figure 14. It does not affect the algorithm except in the case when a conditional send is waiting when a get arrives at the receiver. In this case the process that calls the get should wait on the receiver until the conditional send waiting flag is false. If the conditional send succeeded, and hence executed a put, then the get waiting flag and the conditional send waiting flag should both be false and the actor proceeds through to the third step of the rendezvous. If the conditional send failed, it will have reset the conditional send waiting flag and issued a notifyAll() on the receiver, thus waking up the get and allowing it to properly wait for a put.

The same reasoning also applies to the case when a conditional receive arrives at a receiver before a put.

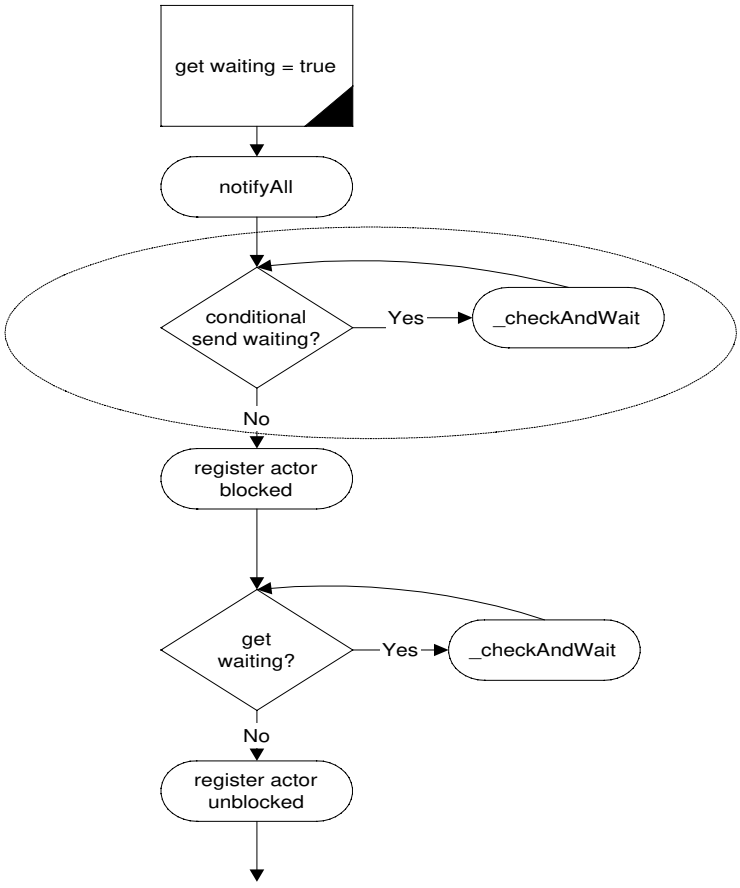


FIGURE 14. Modification of rendezvous algorithm, section 6.4, shown in ellipse

7. Demos and Examples

7.1 DINING PHILOSOPHERS.

This implementation of the Dining Philosophers problem illustrates both time and conditional communication in the CSP domain. Five philosophers are seated at a table with a large bowl of food in the middle. Between each pair of philosophers is one chopstick, and to eat, a philosopher needs both the chopsticks beside him. Each philosopher spends his life in the following cycle: thinks for a while, gets hungry, picks up one of the chopsticks beside him, then the other, eats for a while and puts the chopsticks down on the table again. If a philosopher tries to grab a chopstick but it is already being used by another philosopher, then the philosopher waits until that chopstick becomes available. This implies that no neighboring philosophers can eat at the same time and at most two philosophers can eat at a time.

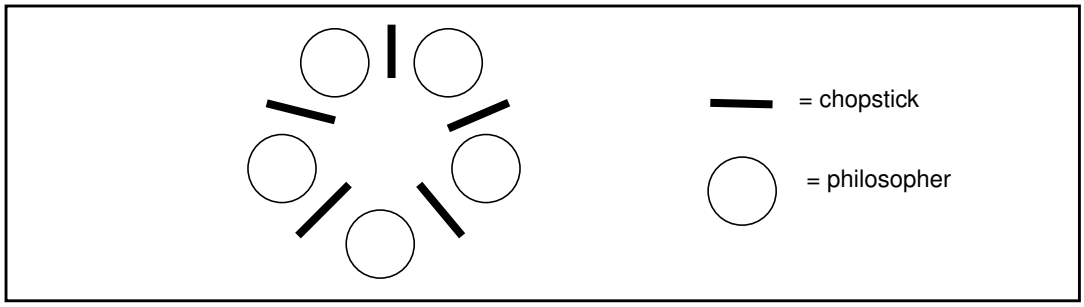


FIGURE 15. Illustration of the Dining Philosophers problem

The Dining Philosophers problem was first dreamt up by Edsger W. Dijkstra in 1965. It is a classic concurrent programming problem that illustrates the two basic properties of concurrent programming:

Liveness. How can we design the program to avoid deadlock, where none of the philosophers can make progress because each is waiting for someone else to do something?

Fairness. How can we design the program to avoid starvation, where one of the philosophers could make progress but does not because others always go first?

This implementation uses an algorithm that lets each philosopher randomly chose which chopstick to pick up first (via a CDO), and all philosophers eat and think at the same rates. Each philosopher and each chopstick are represented by a separate process. Each chopstick has to be ready to be used by either philosopher beside it at any time, hence the use of a CDO. After it is grabbed, it blocks waiting for a message from the philosopher that is using it. After a philosopher grabs both the chopsticks next to him, he eats for a random time. This is represented by calling delay(double) with the random interval to eat for. The same approach is used when a philosopher is thinking. Note that because messages are passed by rendezvous, the blocking of a philosopher when it cannot obtain a chopstick is obtained for free.

This algorithm is fair, as any time a chopstick is not being used, and both philosophers try to use it, they both have an equal chance of succeeding. However this algorithm does not guarantee the absence of deadlock, and if it is let run long enough this will eventually occur. The probability that deadlock occurs sooner increases as the thinking times are decreased relative to the eating times.

7.2 M/M/1

This demo illustrates a simple M/M/1 queue. It has three actors, one representing the arrival of customers, one for the queue holding customers that have arrived and have not yet been served, and the third representing the server. Both the inter-arrival times of customers and the service times at the server are exponentially distributed, which of course is what makes this a M/M/1 queue.

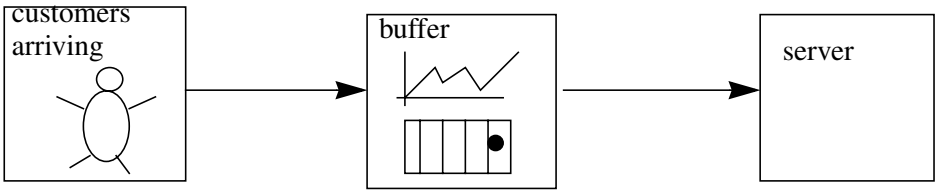


FIGURE 16. Actors involved in M/M/1 demo

This demo makes use of basic rendezvous, conditional rendezvous and time. By varying the rates for the customer arrivals and service times, and varying the length of the buffer, you can see various trade-offs. For example if the buffer length is too short, customers may arrive that cannot be stored and so are missed. Similarly if the service rate is faster than the customer arrival rate, then the server could spend a lot of time idle.

7.3 PAUSING M/M/1

This example demonstrates how pausing and resumption works. The setup is exactly the same as in the M/M/1 demo, except that the thread executing the model calls pause() on the director as soon as the model starts executing. It then waits two seconds, as arbitrary choice, and then calls resume(). The purpose of this demo is to show that the pausing and resuming of a model does not affect the model results, only its rate of progress. The ability to pause and resume a model is primarily intended for the user interface.

7.4 SIEVE OF ERATOSTHENES

This demo illustrates changes to the topology during the execution of a model. It is explained in detail in the section on topology changes, section 8.2.

8. Changes to the Topology during the Execution of a Model

For some models it may be necessary to change the topology of the model during the course of executing the model. This is supported in the CSP domain, but only at specific points of the model execution. In particular, changes to the topology are only allowed at deadlock points.

When the director detects deadlock, real or timed, it then checks if any topology changes have been queued with it. If one or more topology change has been queued, it carries them out and continues. Note that the result of a topology change might remove an otherwise real deadlock by introducing new processes.

8.1 HOW TO WRITE AN ACTOR THAT USES TOPOLOGY CHANGES

The procedure for making a topology change is relatively straightforward. First the actor must create a TopologyChangeRequest object representing the topology change. Second, the request must be queued with the director by calling queueTopologyChange(). If the topology change will not affect any channels or ports the process is communicating with, then the process can proceed. Otherwise the process should delay itself until the next occasion a time deadlock occurs by calling waitForDeadlock(). Then, when the process wakes up again, the director will already have performed the mutation. This is because topology changes get processed when a deadlock is detected, and any queued topology changes are done before waking up delayed processes or advancing time.

The reason for delaying is that it is important that no process be waiting to rendezvous at a receiver in a port affected by the topology change. When a port is affected by a topology change, it is likely that it will abandon its old receivers and create new ones. This will leave the process trying to rendezvous with a dangling receiver, which will eventually cause the model to terminate early. To get around this problem, it is necessary to delay the execution of any processes that may be affected by a rendezvous until the next occasion a time deadlock occurs. For example in the CSPSieve process, each process calls waitForDeadlock() immediately after queueing the mutation.

To create a TopologyChangeRequest, it is necessary to create a subclass that implements the abstract method constructEventQueue(). This is most easily done using an inner class, normally in a private method of the actor. The code in CSPSieve contains an example of this. The reason for using an inner class with a method that creates the topology change is to avoid potential deadlocks. The idea behind avoiding the deadlocks is that the topology changes only happen when the request is processed, which is when the constructEventQueue() method gets invoked. Thus the topology changes are made from *within* the thread that the director is running in, and not the thread running the process that requested the change.

For a more detailed explanation of how changes to the topology are constructed and executed during the execution of a model, and the changes that are allowed, try reading the appropriate section in the Ptolemy II design document[13].

8.2 SIEVE OF ERATOSTHENES EXAMPLE

This example implements the *Sieve of Eratosthenes*. It is an algorithm for generating a list of prime numbers. It originally consists of a source generating integers, and one sieve filtering out all multiples of two. When the end sieve sees a number that it cannot filter, it creates a new sieve to filter out all multiplies of that number. Thus after the sieve filtering out the number two sees the number three, it creates a new sieve that filters out the number three. This then continues with the three sieve

eventually creating a sieve to filter out all multiples of five, and so on. Thus after a while there will be a chain of sieves each filtering out a different prime number. If any number passes through all the sieves and reaches the end with no sieve waiting, it must be another prime and so a new sieve is created for it.

This demo is an example of how changes to the topology can be made in the CSP domain. Each topology change here involves creating a new CSPSieve actor and connecting it to the end of the chain of sieves.

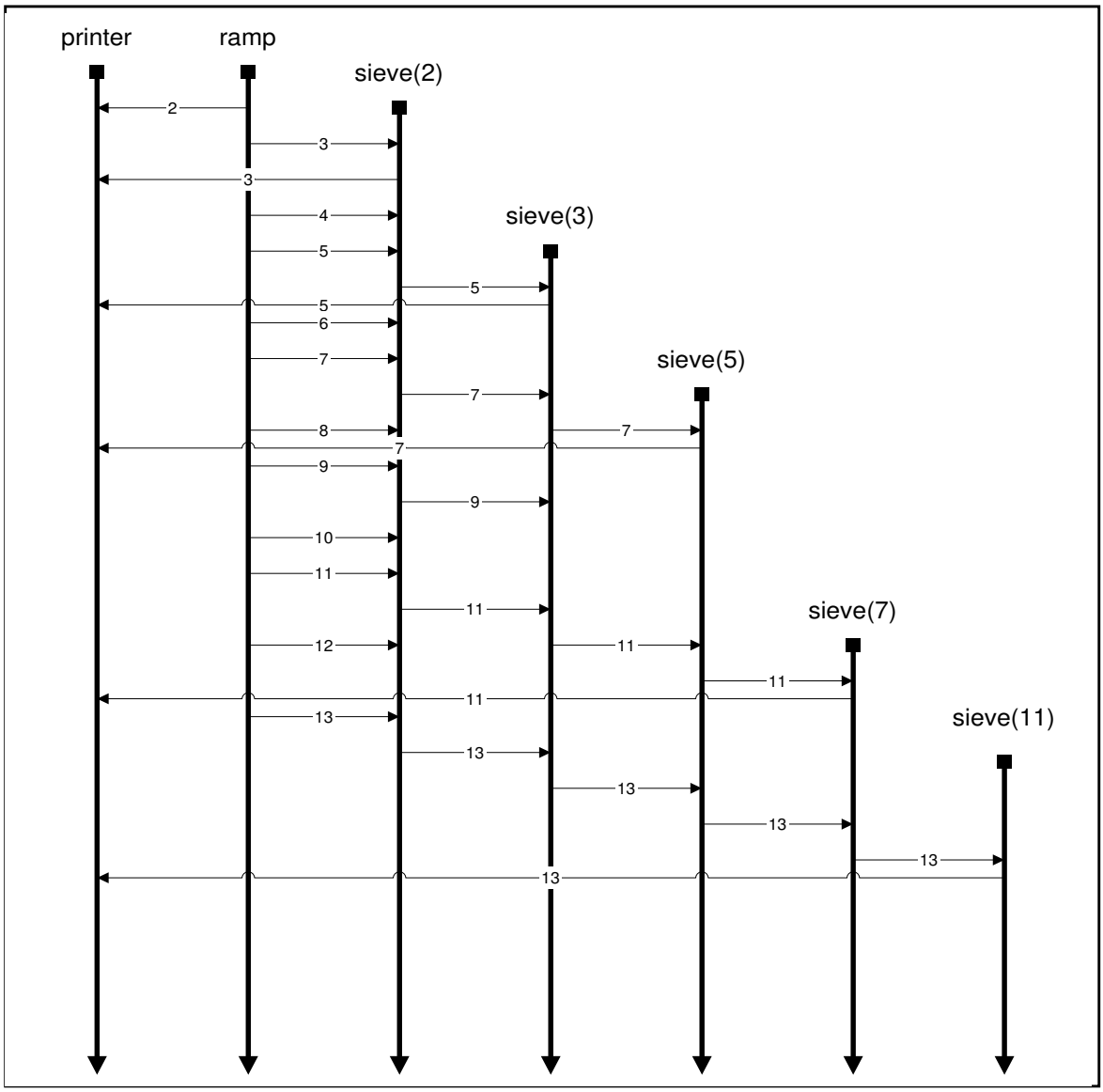


FIGURE 17. Illustration of *Sieve of Eratosthenes* for obtaining first six primes.

9. Composing CSP with other domains

In Ptolemy the mixing of domains is achieved through the use of hierarchy. At any level of the hierarchy, all the actors obey the same semantics (model of computation), but inside any one of these actors there may be another domain using a different model of computation. The composition of CSP with other domains has not yet been fully explored, but a considerable amount of the effort involved in designing the domain was aimed at ensuring smooth interaction between the CSP domain and other domains. In this chapter I have placed some of the thoughts that may be useful in composing CSP with other domains.

9.1 CSP INSIDE ANOTHER DOMAIN

In this case, real deadlock no longer ends the model execution, but instead marks the end of an iteration one level up in the hierarchy. The director transfers any tokens from the inside CSP domain to the outside domain. Control then returns to the outside domain, which continues its execution. Then when fire() is called again on the CSPDirector it transfers any inputs from the outside domain inside and continues until real deadlock is reached again.

The transferring of inputs from the outside domain and inside domain should probably be accomplished using a separate TransferThread object. These threads would simply get a Token from the outside domain, and send it to the channel inside the CSP model. This would be repeated until the thread blocks because there are no more Tokens at the outside level, or when "enough" tokens have been transferred. The director would create one of these threads for each channel that represents an input from one level up in the hierarchy. The director thread will not block as it is not performing the rendezvous directly.

Similarly, when the director is transferring outputs from the CSP model to the model one level up in the hierarchy, it also creates a TransferThread to perform the transfer. If the CSP model wishes to transfer more than one message per iteration up the hierarchy, a CSPBuffer should be placed on each output channel that transfers more than one message. This is to allow the process sending to the output channel to continue after sending the first message.

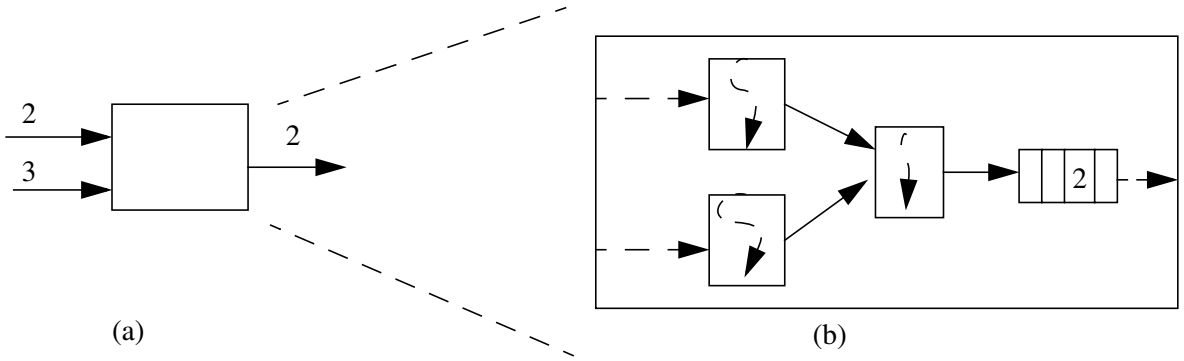


FIGURE 18. Exploded view of what a CSP subsystem might look like inside a composite actor in another domain.

For example if a CSP domain inside a CompositeActor is represented by figure 18(a), then the CSP model inside might have the form shown in (b). The dashed arrows show the transfer of inputs

and outputs between the two levels of the hierarchy.

Each occasion real deadlock occurs, it is guaranteed that no processes are delayed. This allows the time for the CSP model to be set, by the director one level up in the hierarchy, at the start of each iteration. This should make composing CSP with other timed domains reasonably straightforward.

9.1.1 CSP within CSP

The CSP model of computation is not compositional. This means that composing several processes into a single process one level up in the hierarchy may impact the semantics of the model execution. To see this, consider two processes that each simply read an input, then send it on. This is shown in figure 19. If a stream of messages is sent along the input channel of process A, then it will output the same stream of messages on its output channel. No messages are sent along the input channel of process B. If the two processes are then composed as shown by the dashed box, and the composed process reads alternately from the two input channels, then the behavior of the composed process will be to block waiting for a message on the second channel, which is different from that of the two processes separately.

9.2 ANOTHER DOMAIN INSIDE CSP

Recall that each actor in a CSP model is executed by a ProcessThread, as shown in figure 10. Due to the semantics of the CSP model of computation, the inside model is executed in parallel with the other processes. This has implications for the availability of Tokens at the input ports of the model one level down in the hierarchy. If the inside model requires a certain set of Tokens in order to fire, it is up to the director controlling the inside model to ensure this before it executes. This director is also responsible for obtaining any Tokens at each CSPReceiver so that another Token could be sent to the receiver if necessary. This is how the inside domain acquires more than one Token on any given input channel.

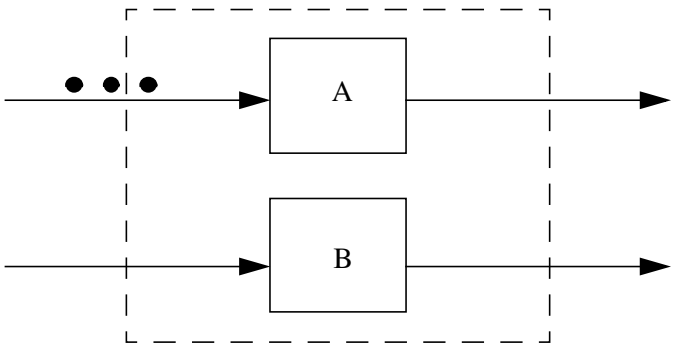


FIGURE 19. Example showing how CSP is not compositional

10. Design decisions

In designing the CSP domain, many design choices had to be made. Below are some of the key design decisions that were made and the motivation for the implementation chosen.

10.1 TIME: DISTRIBUTED RELATIVE TIME VERSUS CENTRALIZED ABSOLUTE TIME

One of the key decisions that had to be made was what model of time to use. The model used was chosen primarily to make composing CSP domains with other timed domains possible. Since each actor only deals with delays relative to the current model time, or at the current model time, then, if no actors are delayed, the current model time can be arbitrarily set. This works well with the notion of an iteration in CSP which is when real deadlock is reached, i.e. when no actors are delayed. Thus the time of a CSP subsystem could be set at the start of each iteration by the director one level up in the hierarchy.

The model also has the added advantage that it is relatively simple and easy to use. The only disadvantage is that time is centralized and so all actions involving time must pass through the director.

10.2 CHOICE OF LOCKS USED AND LOCKING POINTS

The receiver is chosen as the locking point for all communications primarily for scalability. Because the processes involved in a rendezvous lock locally on the receiver involved, the director controlling the model is not directly involved in mediating any rendezvous. If the director were involved, then as the models became larger the performance would suffer as each rendezvous would have to be carried out through the director. The receiver is a natural point for storing the flags involved in controlling a conditional communication. Note that a rendezvous is completely separate from the notion of time in the domain.

There are three primary lock types in use in the CSP domain: the director lock, of which there is only one, a lock for each receiver, and an internal lock hidden inside each actor. The hidden lock simply takes the place of locking on the actor for internal control mechanisms. The use of each of these locks should not be visible when using the domain. The decision to use an internal lock for controlling access to methods of CSPActor was made to avoid using any lock that the code in a user written actor might use. In particular, the actor code should be able to *lock on itself.* If we had chosen to lock onto the actor itself, as opposed to a hidden lock, then the model could deadlock if the actor code synchronized on itself.

10.3 MAKING ALL THE COMMUNICATION MECHANISMS SYMMETRIC

Aside from the fact that a Token is transferred or received in a rendezvous, the two actions are symmetric, so I felt that the locking algorithms should also be. This also has the advantage of making the algorithm easier to understand and less prone to unintended deadlocks as there are fewer interactions to consider. Similarly the choice of making the algorithm used in the guarded communication threads symmetric is made to keep it as simple, understandable and as robust as possible.

10.4 CONDITIONAL COMMUNICATION MECHANISMS UPON RENDEZVOUS

The reason for building the conditional communication mechanism upon the rendezvous mecha-

nism is that it is logically clearer what is happening when it is separated into two steps: first decide which branch will rendezvous, then do the rendezvous. This also enables tracking down and removing situations where false deadlocks could arise.

10.5 POINTS IN THE MODEL WHEN CHANGES TO THE TOPOLOGY ARE ALLOWED

The options considered for when to allow changes to the topology are either at deadlock points, or as soon as the model can be paused. The reason for choosing to allow changes to the topology only at deadlock points is mainly that these points are intrinsic to the nature of the model. The state of the model is well defined at these points: all processes are either blocked trying to communicate or are delayed waiting for time to advance. For any execution of a model, the times at which time deadlocks occur are the times at which topology changes may occur. This allows for a process to be created when another process reaches some state, and the two processes will be continuing from the same model time.

Pausing and resuming a model does not affect the outcome of a particular model run, only the rate of progress. Thus if changes to the topology were allowed to happen immediately (as soon as the model is able to pause), this would result in a new nondeterminism being introduced into the model. For CSP we wish to keep all nondeterminism the result of the conditional communication constructs, so topology changes are only allowed at deadlocks.

11. Conclusion and Future Work

The CSP domain in Ptolemy II has been implemented using the concurrency support built into Java. It builds upon the low level support Java offers to allow the user to design concurrent systems at a much higher level of abstraction. A notion of time has been added to the classical CSP model to enable modeling of systems where time is relevant, in particular embedded systems. Finally, the CSP domain allows the topology of a model to change during execution while still maintaining a consistent state.

The composition of CSP with other domains is important for heterogeneous modeling of systems. In particular it is envisioned that the CSP domain will be hierarchically composed in models where resource contention is a major concern. Some examples include embedded systems where a number of functions share the same CPU, or in modeling client/server architectures.

The hierarchical composition of the CSP domain with other domains in Ptolemy II has not yet been fully explored. However, much of the effort in designing the CSP domain was devoted to ensuring that the CSP domain could be successfully composed with other domains. It should make for some very interesting research defining and exploring the semantics of these interactions. It is regrettable that I did not have enough time to start exploring this area. I believe the design and the algorithms used in the domain are sufficiently adaptable/clear that the domain should be fairly easy to extend or modify if necessary.

References

[1] G. R. Andrews, *Concurrent Programming - Principles and Practice*, Addison-Wesley, 1991.

[2] A. Burns, *Programming in OCCAM 2*, Addison-Wesley, 1988.

[3] P. H. J. van Eijk, C. A. Vissers, M. Diaz, *The formal description technique LOTOS*, Elsevier Science, B.V., 1989. (http://wwwtios.cs.utwente.nl/lotos)

[4] M. Fowler and K. Scott, *UML Distilled*, Addison-Wesley, 1997.

[5] A.J.C. van Gemund, *"Performance Prediction of Parallel Processing Systems: The PAMELA Methodology,"* Proc. 7th Int. Conf. on Supercomputing, pages 418-327, Tokyo, July 1993.

[6] M. G. Hinchey and S. A. Jarvis, *Concurrent Systems: Formal Developments in CSP*, McGraw-Hill, 1995.

[7] C. A. R. Hoare, *"Communicating Sequential Processes,"* Communications of the ACM, Vol. 21, No. 8, August 1978.

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

[9] G. Kahn, *"The Semantics of a Simple Language for Parallel Programming,"* Proc. of the IFIP Congress 74, North-Holand Publishing Co., 1974.

[10] D. Lea, *Concurrent Programming in Java™*, Addison-Wesley, MA, 1997.

[11] E. A. Lee and T. M. Parks, *"Dataflow Process Networks,"* Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801, May, 1995. (http://ptolemy.eecs.berkeley.edu/papers/processNets)

[12] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. **Ph.D. Dissertation**. EECS department, University of California, CA 94720, December 1995. http://ptolemy.eecs.berkeley.edu/papers/parksThesis)

[13] The Ptolemy Project, *PtolemyII*, http://ptolemy.eecs.berkeley.edu/ptolemyII.