Chapter 6 from: C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng "Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)," Technical Memorandum UCB/ERL M04/17, University of California, Berkeley, CA USA 94720, June 24, 2004.

6

CSP Domain

Author: *Neil Smyth*
Contributors: *Elaine Cheong*
John S. Davis II
Bilung Lee
Steve Neuendorffer

6.1 Introduction

The communicating sequential processes (CSP) domain in Ptolemy II models a system as a network of sequential processes that communicate by passing messages synchronously through channels. If a process is ready to send a message, it blocks until the receiving process is ready to accept the message. Similarly if a process is ready to accept a message, it blocks until the sending process is ready to send the message. 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.

The CSP domain is based on the model of computation (MoC) first proposed by Hoare [59][60] in 1978. In this MoC, a system is modeled as a network of processes communicate solely by passing messages through unidirectional channels. The transfer of messages between processes is via *rendezvous*, which means both the sending and receiving of 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 Gregory Andrews' book on concurrent programming [7], which refers to rendezvous-based message passing as *synchronous message passing*.

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 [57]. It might thus be more accurate to refer to the domain using our model of computation as the "Timed CSP" domain, but since it can be used with and without time, it is simply referred to as the CSP

domain.

6.2 Properties of the CSP Domain

At the core of CSP communication semantics are two fundamental ideas. First is the notion of atomic communication and second is the notion of nondeterministic choice. It is worth mentioning a related model of computation known as the calculus of communicating systems (CCS) that was independently developed by Robin Milner in 1980 [104]. The communication semantics of CSP are identical to those of CCS.

6.2.1 Atomic Communication: Rendezvous

Atomic communication is carried out via rendezvous and implies that the sending and receiving of a message occur simultaneously. During rendezvous both the sending and receiving processes block until the other side is ready to communicate; the acts of sending and receiving are indistinguishable activities since one can not happen without the other. A real world analogy to rendezvous can be found in telephone communications (without answering machines). Both the caller and callee must be simultaneously present for a phone conversation to occur. Figure 6.1 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.

6.2.2 Choice: Nondeterministic Rendezvous

Nondeterministic choice provides processes with the ability to randomly select between a set of possible atomic communications. We refer to this ability as nondeterministic rendezvous and herein lies much of the expressiveness of the CSP model of computation. The CSP domain implements nondeterministic rendezvous via *guarded communication statements.* A guarded communication state-

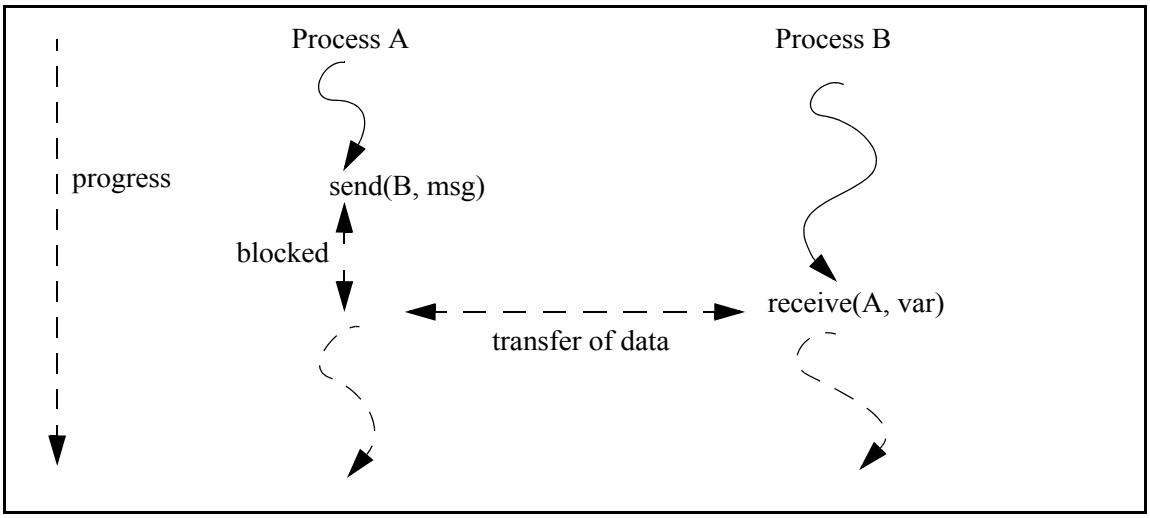

FIGURE 6.1. Illustrating how processes block waiting to rendezvous

ment has the form

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

The *guard* is only allowed to reference local variables, and its evaluation cannot change 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. *Statements* 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** (conditional if) and **CDO** (conditional do). These are analogous to the *if* and *while* statements in most programming languages. 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.

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. Once the succeeding communication is carried out, its 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.

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 non-deterministically. 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.

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 6.2. Note that in this case both guards can never be simultaneously false so this process will execute the CDO forever.

6.2.3 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.

6.2.4 Time

In the CSP domain, *time* is centralized. That is, all processes in a model share the same time, referred to as the *current model time*. Each process can only choose to *delay* itself for some period relative to the current model time, or a process can wait for time deadlock to occur at the current model time. In 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.

A process can also choose to delay its execution until the next occasion a time deadlock is reached. The process resumes execution at the model time at which it began its delay. This allows a model to several sequences of actions at the same model time. The next occasion time deadlock is reached, any

```
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 6.2. Example of how a CDO might be used in a buffer.

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 6.5.2.

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; time can only change 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. The choice for modeling time was in part influenced by Pamela [44], a run time library that is used to model parallel programs.

6.2.5 Differences from Original CSP Model as Proposed by Hoare

The model of computation used by the CSP domain differs from the original CSP [59] 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 [57]. Second, as mentioned in section 6.2.2, 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 [60].

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 by Hoare. For example, the languages Occam [23] and Lotos [36] 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 inter-operability 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.

6.3 Using CSP

There are two basic issues that must be addressed when using the CSP domain:

- Unconditional vs. conditional rendezvous
- Time

6.3.1 Unconditional vs. Conditional Rendezvous

The basic communication statements send() and get() correspond to rendezvous communication in the CSP domain. The fact that a rendezvous is occurring on every communication is transparent to the actor code due to the domain framework. However, this rendezvous is *unconditional* (deterministic); an actor can only attempt to communicate on one port at a time. To realize the full power of the CSP domain, which allows non-deterministic rendezvous, it is necessary to write custom actors that use the *conditional* communication constructs in the CSPActor base class. 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.

2) Pass the branches to the chooseBranch() method in CSPActor. This method evaluates the guards, 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.

A sample template for executing a conditional communication is shown in figure 6.3. This template corresponds to the CDO (conditional do) construct in CSP, described in section 6.2.2. 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 fifth argument (for ConditionalSend) contains the token to be sent. 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 because once the guard is false, the token in a ConditionalSend is never referenced.

The code for using a CIF (conditional if) is similar to that in figure 6.3 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.

Figure 6.4 shows some actual code based on the template above that implements a buffer process. This process repeatedly rendezvous on its input port and its output port, buffering the data if the reading process is not yet ready for the writing process. It is worth pointing out that if most channels in a

```
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 {
      // error
   }
}
```

FIGURE 6.3. Template for executing a CDO construct.

model are buffered in this way, it may be easier to create the model in the PN domain, where every channel implicitly has an unbounded buffer.

6.3.2 Time

The CSP domain does not currently use the fireAt() mechanism to model time. If an actor wishes be delayed a certain amount of time during execution of the model, it must derive from CSPActor. 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() and waitForDeadlock(). 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()

As far as each process is concerned, time can only increase while it is blocked waiting to rendezvous or when it is delayed. A process can be aware of the current model time, but it should only affect the model time by delaying its execution, thus forcing time to advance. The method setCurrentTime() should never be called from a process. However, if no processes are delayed, it is possible to set the model time by calling the setCurrentTime() method of the director. However, this method is present only for composing CSP with other domains.

By default every model in the CSP domain is timed. To use CSP without a notion of time, simply do not use the delay() method. The infrastructure supporting time does not affect the model execution if this method is not used. For more information about the semantics of Timed CSP models, see section 6.2.4

```
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 6.4. Code used to implement the buffer process described in figure 6.3.

6.4 The CSP Software Architecture

6.4.1 Class Structure

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 CSP semantics to a model. The CSP domain associates each channel with exactly one receiver, which is located at the receiving end of the channel. Thus any process that sends or receives to any channel will rendezvous at a CSPReceiver. Figure 6.5 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.

CSPDirector: This gives CSP semantics to a model. 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: This ensures that communication of messages between processes is via rendezvous.

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

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

6.4.2 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.

Figure 6.6 shows the code executed by the ProcessThread class. Note that it makes no assumption

```
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 6.6. Sequence of steps involved in setting up and controlling the model.

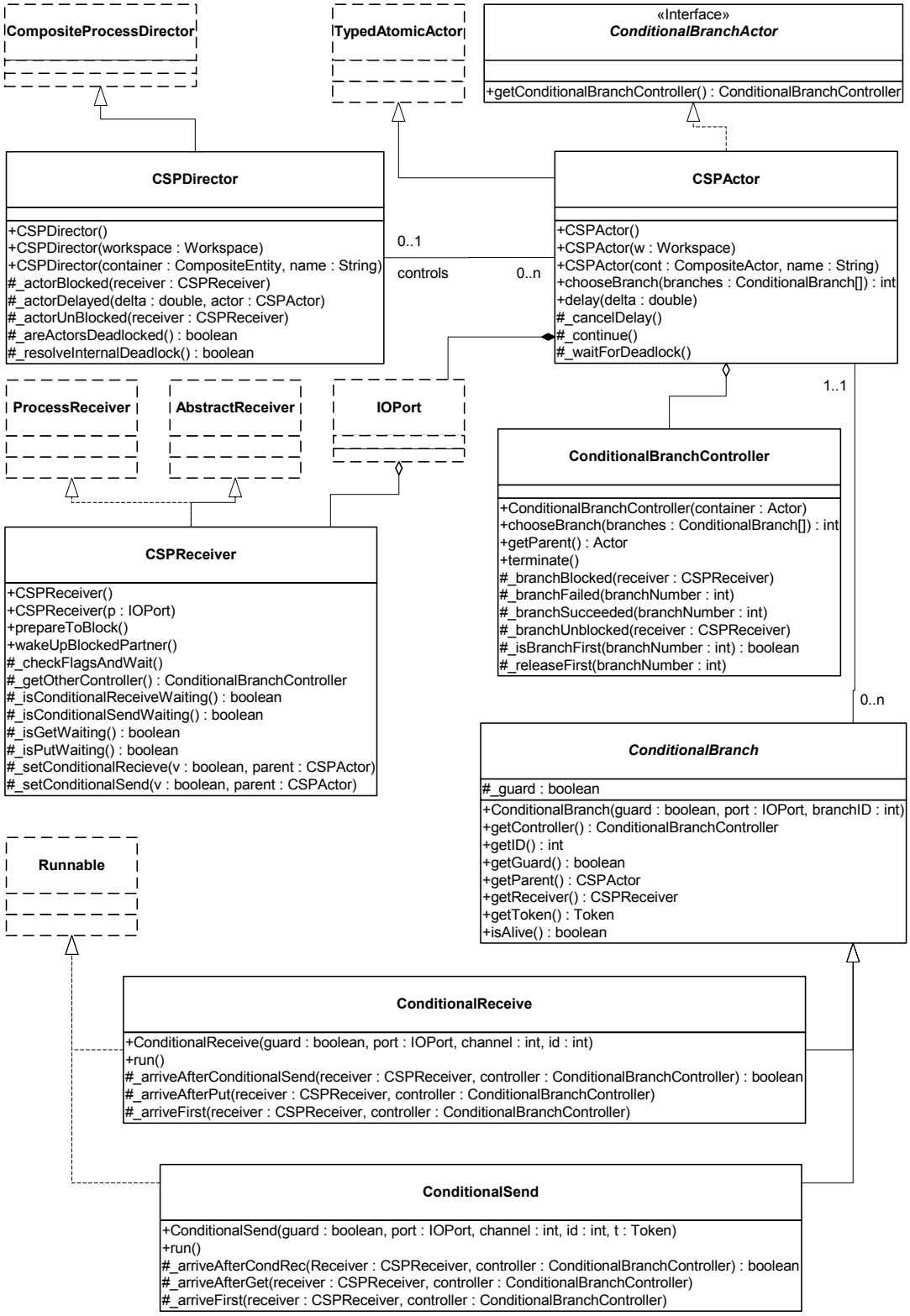

FIGURE 6.5. Static structure diagram for classes in the CSP kernel.

about the actor it is executing, so it can execute any domain-polymorphic actor as well as CSP domain-specific actors. In fact, an actor from any other domain that does not rely on the specifics of its parent domain can be executed in the CSP domain by the ProcessThread.

6.4.3 Detecting deadlocks:

For deadlock detection, the director maintains three counters:

- 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 processes 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 awakened 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 increases 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 6.7. 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.

6.4.4 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 by being terminated *early* by a TerminateProcessException. 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 6.7) representing the process to exit its while loop and execute its finally clause. The finally clause calls wrapup() on the actor it represents and decreases the count of active processes in the director; 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 be impossible. It should rarely be called.

6.4.5 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 be called similarly 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.

6.5 Example CSP Applications

Several example applications have been developed which serve to illustrate the modeling capabilities of the CSP model of computation in Ptolemy II. Each demonstration incorporates several features of CSP and the general Ptolemy II framework. The applications are described here, but not the code. See the directory $PTII/ptolemy/domains/csp/demo for the code.

The first demonstration, *dining philosophers*, serves as a natural example of core CSP communication semantics. This demonstration models nondeterministic resource contention, e.g., five philosophers randomly accessing chopstick resources. Nondeterministic rendezvous serves as a natural modeling tool for this example. The second example, *hardware bus contention*, models deterministic resource contention in the context of time. As will be shown, the determinacy of this demonstration constrains the natural nondeterminacy of the CSP semantics and results in difficulties. Fortunately these difficulties can be smoothly circumvented by the timing model that has been integrated into the CSP domain.

6.5.1 Dining Philosophers

Nondeterministic Resource Contention. 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 that 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.

The dining philosophers problem was first proposed 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

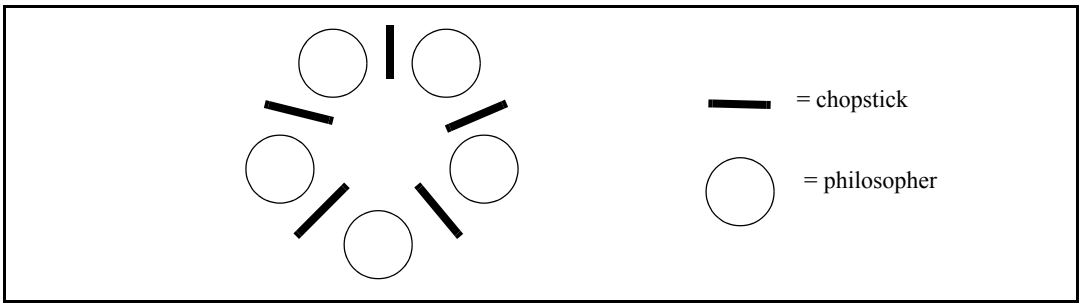

FIGURE 6.8. Illustration of the dining philosophers problem.

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 is 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() 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.

6.5.2 Hardware Bus Contention

Deterministic Resource Contention. This demonstration consists of a controller, *N* processors and a memory block, as shown in Figure 6.9. At randomly selected points in time, each processor requests permission from the controller to access the memory block. The processors each have priorities associated with them and in cases where there are simultaneous memory access requests, the controller grants permission to the processor with the highest priority. Due to the atomic nature of rendezvous, it is impossible for the controller to check priorities of incoming requests while requests are occurring. To overcome this difficulty, an alarm is employed. The alarm is started by the controller immediately following the first request for memory access at a given instant in time. It is awakened when a delay block occurs to indicate to the controller that no more memory requests will occur at the given point in time. Hence, the alarm uses CSP's notion of delay blocking to make deterministic an inherently non-deterministic activity.

6.6 Technical Details

6.6.1 Rendezvous Algorithm

In CSP, the locking point for all communication between processes is the receiver. Any time a pro-

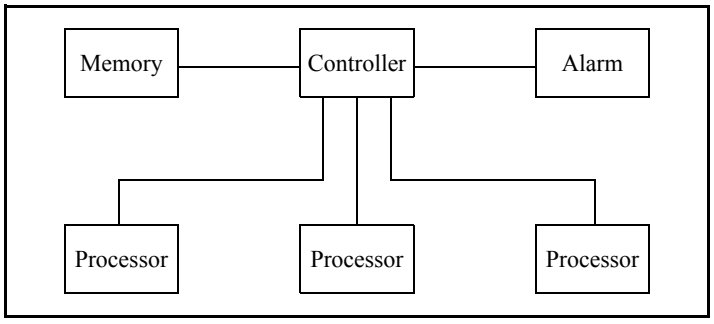

FIGURE 6.9. Illustration of the Hardware Bus Contention example.

cess wishes to send or receive, it must first acquire the lock for the receiver associated with its communication channel. 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 number of deadlock situations that could arise and also makes the interaction between processes more understandable and easier to explain. The algorithm for get() is shown in figure 6.10. The algorithm for 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.

When a get() arrives at a receiver, a put() is either already waiting to rendezvous or it is not. 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 *getWaiting* 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. This situation is avoided by our design.

6.6.2 Conditional Communication Algorithm

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

Built on top of rendezvous:
When a conditional construct has more than one enabled branch (the 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 6.6.1, with the exception of one special case, which is described in section 6.6.3. 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.

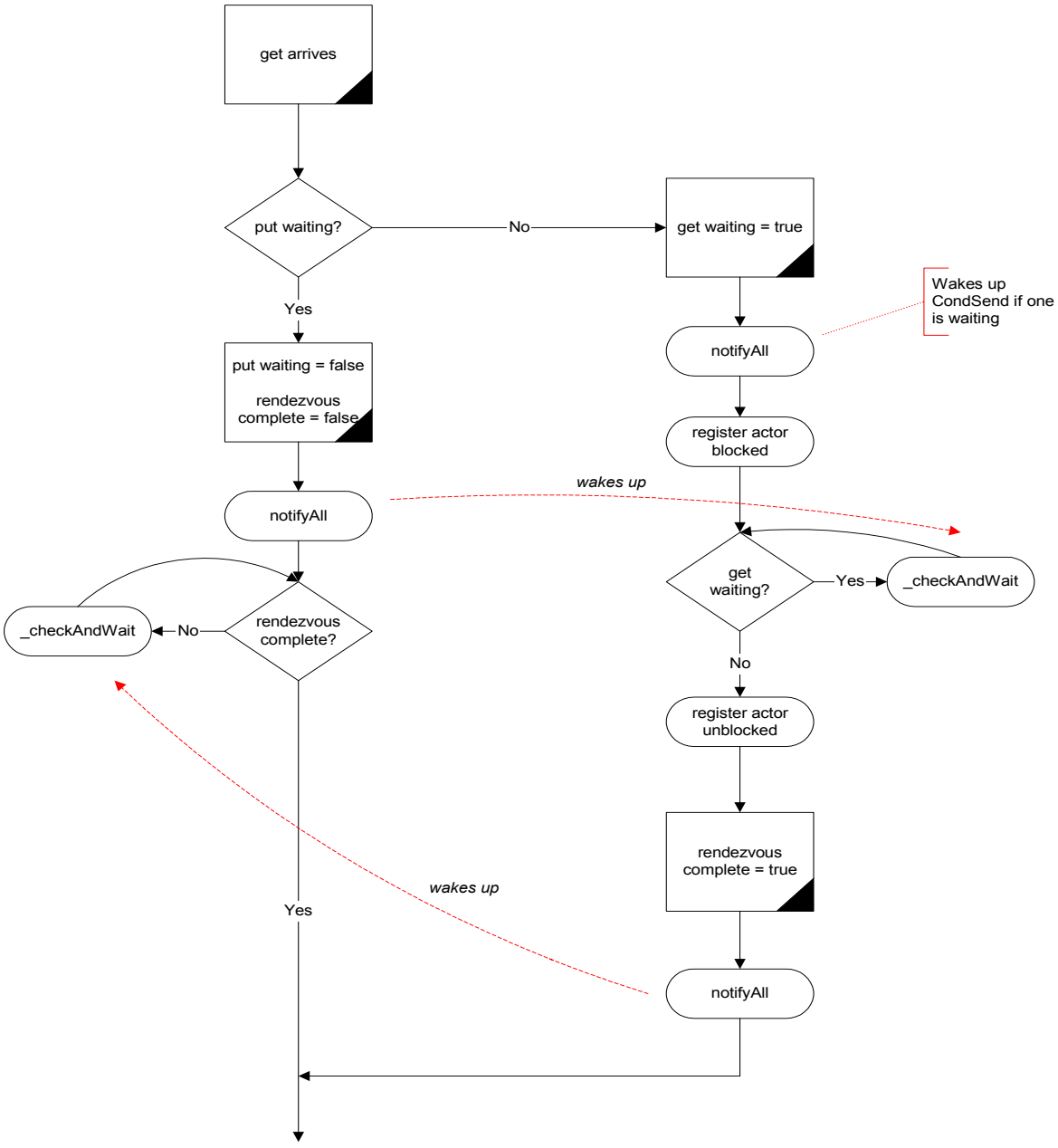

FIGURE 6.10. Rendezvous algorithm.

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 performs a rendezvous successfully.

Algorithm used by each branch:
Similar to the approach followed in 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 6.12. 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

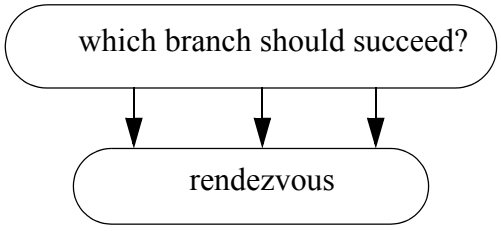

FIGURE 6.11. Conceptual view of how conditional communication is built on top of rendezvous.

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 whether 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

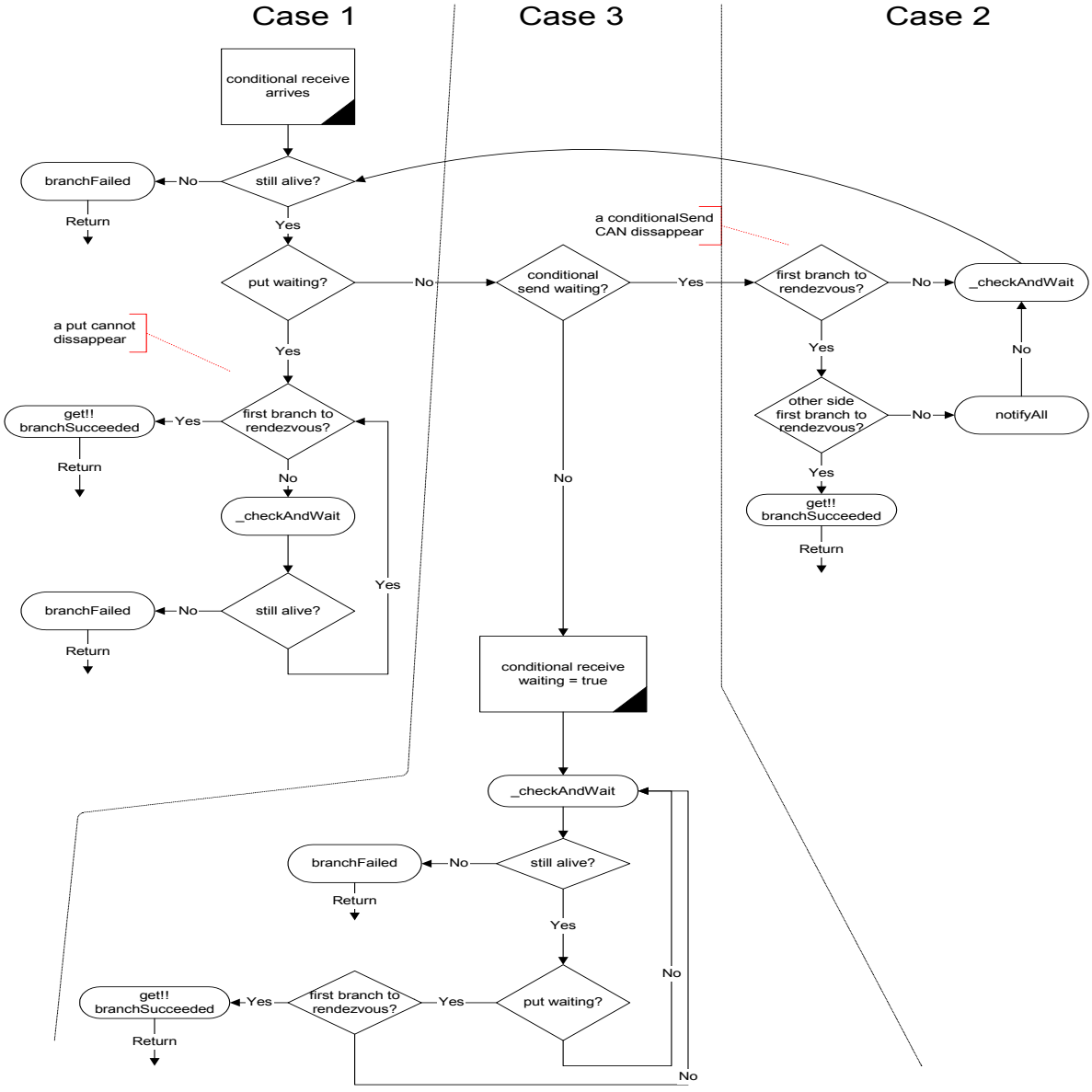

FIGURE 6.12. Algorithm used to determine if a conditional rendezvous branch succeeds or fails

and executes a get. If it is not the first, it waits on the receiver and tries again.

6.6.3 Modification of Rendezvous Algorithm

Consider the case when a conditional send arrives before a get. If all the branches in the conditional communication that the conditional send is a part of are blocked, then the process will register itself as blocked with the director. When 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 the algorithm is shown in the dashed ellipse in figure 6.13. 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.

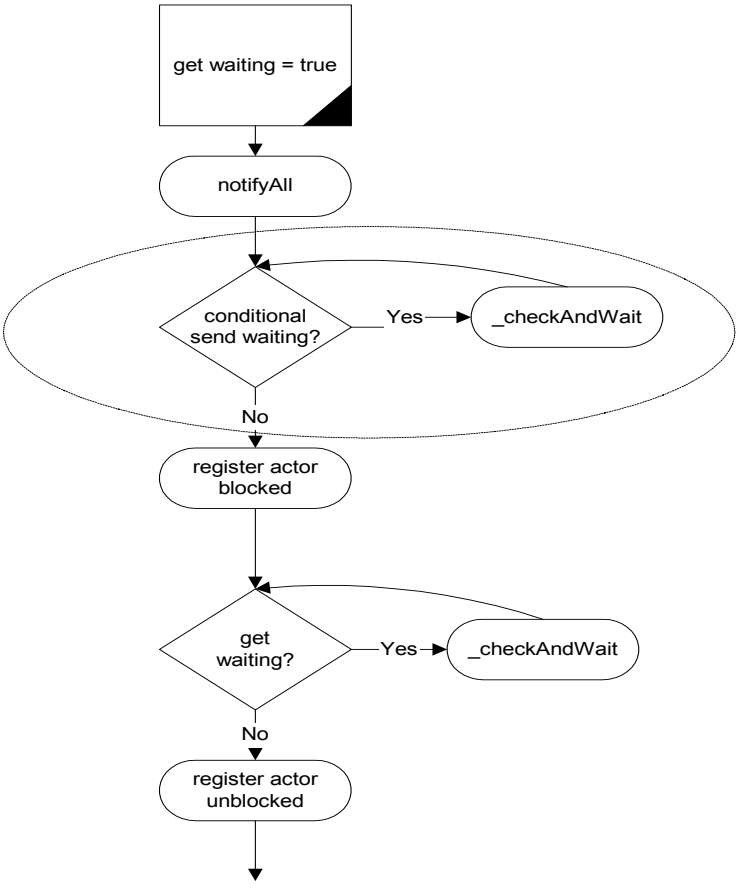

FIGURE 6.13. Modification of rendezvous algorithm, section 6.6.3, shown in ellipse.