Representing Communicating Software to Derive System Behavior and Deadlock-Free Software

Debra S. Lane

Technical Report No. 87-27

University of Calfornia at Irvine

Department of Information and Computer Science

November 2, 1987

ABSTRACT

A great difficulty in building distributed systems lies in being able to predict what the systems behavior will be. A distributed or communicating system is defined here to be one in in which the hardware consists of a set of processors each with their own memory, connected by some communication medium (there is no shared memory), and the software is assumed to be of the CSP (Hoare's Communicating Sequential Processes) type.

In the past few years some theories have been proposed to model features of communicating systems. Milner's Calculus of communicating Systems (CCS), Winskel's Synchronization Trees (ST), Hennessy's Acceptance Trees (AT), and Hoare and Brookes's theory of communicating processes are examples of formal models of such systems. All of these models concentrate on modelling observable properties of a system.

Event Dependency Trees (EDT) is a new representation of communicating systems that models the time dependent nature of such systems. None of the representations mentioned above explicitly represent time but time is precisely the factor that introduces so much variability and complexity into such software and systems. EDT provides a representation based on trees and a set of operations over the EDT trees that can be used to produce deadlock–free software. The model supplies potentially important information for the design and construction of distributed, parallel software systems.

Representing Communicating Software to Derive System Behavior and Deadlock-Free Software

Introduction

A great difficulty in building distributed systems lies in being able to predict what the system behavior will be. A distributed or communicating system is defined here to be one in which the hardware consists of a set of processors each with their own memory, connected by some communication medium (there is no shared memory), and the software is assumed to be of the CSP (Hoare's Communicating Sequential Processes) type. The problem is that while it is easy to understand how each process behaves in and of itself, it is nearly impossible to predict all the ways in which the processes will interact and influence each other's execution. It is necessary to understand their interaction in order to determine how the system behaves (so that one might convince oneself or others that the system performs as intended).

In the past few years some theories have been proposed to model features of communicating systems. Milner's Calculus of Communicating Systems (CCS) [MILN80], Winskel's Synchronization Trees (ST) [WINS84], Hennessy's Acceptance Trees (AT) [HENN85B], and Hoare and Brookes's theory of communicating processes [BROO84] are examples of formal models of such systems. All of these models concentrate on modelling observable properties of a system.

This paper presents a new representation of communicating systems called Event Dependency Trees (EDT) [LANE87] that models the time dependent nature of such systems. None of the representations mentioned above explicitly represent time but time is precisely the factor that introduces so much variability and complexity into such software and systems. Many models in computer science

assume that events occur instantaneously, but here it is assumed that every event occurs with a certain time delay represented explicitly by an event name and a variable for the time delay. Communication events are important because that is how processes interact. Events preceding the communication events, even if they are only executions of sequential pieces of code, are also very important, however, because they determine the exact manner in which the communication events will occur.

Besides modelling time explicitly, EDT differs from CCS, ST, and AT in its representation of system behavior. Both CCS and ST represent system behavior as interleavings of events. The combine tree operation in those models produces the set of interleavings. AT represents the system as a state–transition graph. The tree combine operation in AT takes two state–transition graphs and produces a larger one. In EDT, the system behavior is represented as a partial ordering of events. The combine tree operation in EDT produces the partial ordering of events in a way that indicates how particular sets of events contend with each other to produce the various execution paths.

EDT show the right amount of information about system behavior, not too much as in an interleaving representation, and not too little as in a state–transition model. It is possible to identify each execution path by its *unique* event ordering. In interleaving many event orderings produce the same execution path because many times it is irrelevant that some event occurred before or after another since they don't influence each other's execution. EDT shows exactly those events that influence each other's execution and also those that are not related.

CCS, ST, and AT all show the possible execution paths but indicate only that they arise because of nondeterminism. What is the source of such nondeterminism? There are two ways in which nondeterminism arises in such systems: (1) through the use of guarded commands, and (2) through the use of the communication

constructs. EDT models the nondeterminism that arises through the use of communication constructs in CSP–type languages.

Because of limited space this paper tries to provide an intuitive feel for the structure of Event Dependency Trees, their operations, how they model time dependent behavior (i.e., their explicit representation of time and depiction of system behavior), and how they can be used to detect deadlock. In fact, one type of deadlock will never be manifest in the representation of the system because it can be detected from the structure of the trees as the overall behavior is derived.

Event Dependency Trees

In EDT processes are represented as trees where the nodes of a tree represent system states and the arcs represent the execution of system events. An event is one of three types: (1) execution: represents the execution of a sequential piece of code (with no communication constructs), (2) communication: represents the execution of a message passing construct, or (3) the null event. Communication events are further subdivided into send, receive, and synchronized communication events. In addition, each event has an associated time delay, represented by some variable such as t.

The following notation is used:

1) $\overrightarrow{e[t]}$ denotes a sending communication event that takes time t.
2) $\overleftarrow{e[t]}$ denotes a receiving communication event that takes time t.
3) $\overleftrightarrow{e[t]}$ denotes a synchronized communication event that takes time t.
4) $e[t]$ denotes an execution event that takes time t.
5) τ_0 denotes the null tree, which is also the null event.

These are the only events that can occur in EDTs. Using this model, all portions of the computation that take time are accounted for.

Labelling trees is subject to some restrictions, which are not described here. However, note that each event has a name e, a time t, and a type that is in the set $\{exec, send, recv, sync, null\}$. The name of the null event, which is also the null tree, is ε or the empty string, and the time of the null tree is 0. The functions *name*, *type*, and *time* when applied to an event, return the respective information about that event.

Two operations are defined on trees: a prefix operation that allows a tree to be prefixed by an event producing a new tree (prefixing an event to the null tree results in a tree with a single arc labelled by the new event); and a combine operation that takes two trees and produces a new tree. The combine operation is a very important one in that it preserves the relevant information that indicates how execution paths arise as a function of event orderings. Many preliminary definitions and functions are needed to define the combine operation.

First the notion of *matching communication events*, which occurs between trees, not within a tree, is defined. Communication events are important because they are the only way that processes interact.

Definition 2.2. *Let $\mathcal{A}$ be a set of events. $\forall \alpha, \beta \in \mathcal{A}$, α and β are matching communication events, denoted $\alpha \overset{m.c.e}{=} \beta$ if and only if*

i) $name(\alpha) = name(\beta)$,

ii) $type(\alpha) = send$ *and* $type(\beta) \in \{recv, sync\}$ *OR* $type(\alpha) \in \{recv, sync\}$ *and* $type(\beta) = send$.

Thus, matching communication events are two events with the same event name in which either (i) one is a receiving communication event and one is a sending communication event, e.g., $c[\overleftarrow{t_2}]$ and $c[\overrightarrow{t_1}]$, or (ii) one is a synchronized communication event and one is a sending communication event, e.g., $c[\overleftrightarrow{t_1}]$ and $c[\overrightarrow{t_2}]$.

Now, given two arbitrary trees, it is necessary to determine whether or not they have matching communication events and if they do, to identify them.

Definition 2.3. *$\mathcal{L}_\tau$ is the set of all event labels in tree τ.*

Next, a function $\mathcal{COMM}$ is defined that takes an EDT and maps it to a list of the communication events it contains.

Definition 2.4. *Let τ be some EDT. $\mathcal{COMM}(\tau) = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ where $\forall i \in \{1, \ldots, n\}, \alpha_i \in \mathcal{L}_\tau$, $type(\alpha_i) \in \{send, recv, sync\}$ and there does not exist any $\beta \in (\mathcal{L}_\tau \setminus \{\alpha_1, \ldots, \alpha_n\}) \ni type(\beta) \in \{send, recv, sync\}$.*

Two trees, τ, μ, having matching communication events is denoted $\mathcal{COMM}(\tau) \ominus \mathcal{COMM}(\mu)$, stated formally below. For the following definitions, let $\mathcal{EDT}$ be a set of EDTs.

Definition 2.5. *Let $\tau, \mu \in \mathcal{EDT}$, and $\mathcal{COMM}(\tau) = (\alpha_1, \ldots, \alpha_n)$, $\mathcal{COMM}(\mu) = (\beta_1, \ldots, \beta_m)$. If $\exists i \in \{1, \ldots, n\}$ and $\exists j \in \{1, \ldots, m\} \ni \alpha_i \overset{mce}{=} \beta_j$, then $\mathcal{COMM}(\tau) \ominus \mathcal{COMM}(\mu)$.*

$\mathcal{MATCH}$ is a function that maps two trees to a list of all their matching communication events. If $\mathcal{MATCH}$ contains more than one pair of matching communication events, then if the portion of the multiple pairs in one tree occurs in a chain, then the respective portion in the other tree must also occur in a chain. There can not be branch nodes ocurring between one portion of the pair in one tree and not in the other. The reason is that the resulting tree will contain a deadlock. This is discussed in more detail later.

Definition 2.6. *Let $\tau, \mu \in \mathcal{EDT} \ni \mathcal{COMM}(\tau) \ominus \mathcal{COMM}(\mu)$. $\mathcal{MATCH}(\tau, \mu) = ((\alpha_{i_1}, \ldots, \alpha_{i_k}), (\beta_{j_1}, \ldots, \beta_{j_k}))$ where $k \in \{1, \ldots, min\{n, m\}\}$ and $\alpha_{i_l} \overset{mce}{=} \beta_{j_l}$.*

There are two more pieces of information that will be needed: the length of the path from the root node to some designated event in the tree, and a "route"

indicating which branches to take to arrive at the designated event, beginning at the root of the tree.

Definition 2.7. *Let* $\tau \in \mathcal{EDT}$, $\alpha \in \mathcal{L}_\tau$. $\mathcal{PATH}(\tau, \alpha) = n$, *where* $n \in \mathcal{NAT}$ *is the length of the path from the root node to* α.

Definition 2.8. *Let* $\tau \in \mathcal{EDT}$, $\alpha \in \mathcal{L}_\tau$, r *the root node, and* ε *the empty string.* $\forall \alpha, \forall s \in \mathcal{NAT}^*$, *and* $\forall i \in \mathcal{NAT}$,

i) $\mathcal{DEST}(\alpha, \varepsilon) = \alpha$,

ii) $\mathcal{DEST}(\alpha, si) =$ *the* i*th child of* $\mathcal{DEST}(\alpha, s)$.

$\mathcal{DEST}$ is not defined in some cases (e.g., the third child of a node with only two children).

Definition 2.9. *Let* $\tau \in \mathcal{EDT}$, $\alpha \in \mathcal{L}_\tau$, *and* r *the root node.* $\mathcal{ROUTE}(\tau, \alpha) = s \ni s \in \mathcal{NAT}^*$ *and* $\mathcal{DEST}(\alpha, s) = r$.

The combine operation can be thought of as taking two concurrent processes and showing how they interact and affect each other. If the two processes do not exchange information (i.e., they don't send messages to each other), then they will not affect each other and the corresponding trees that represent them will be denoted as a tuple (of trees) called a pseudo tree. Each pseudo tree is actually a forest of trees. Two trees will be combined into a single (new) tree when they have matching communication events. The tree that contains the sending communication event will be referred to as the *active tree* and the tree that contains the other event in the matching communication events pair, the *passive tree.*

Rather than give the formal definition since it is quite lengthy, the combine operation is defined pictorially. Figure 1 shows all the cases that arise when combining two trees that contain matching communication events. Each tree is broken into a subtree prefixed by an arc. Selective

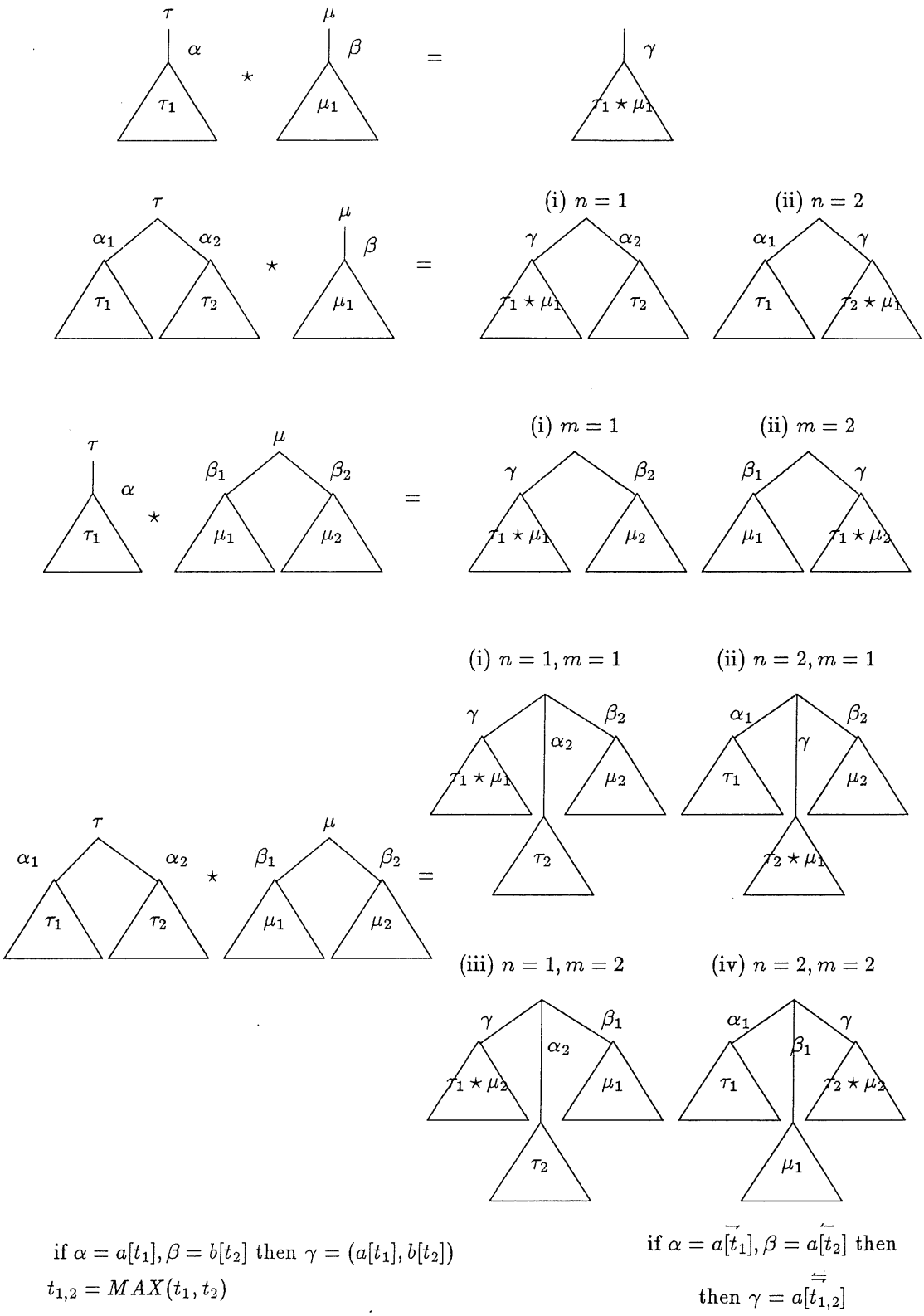


Figure 1

Combining Trees

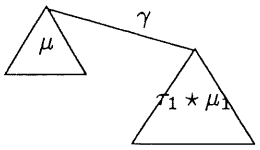
subtrees are recursively combined. Referring to Figure 1, assume δ_1 and δ_2 are the pair of matching communication events such that $\delta_1 \in \mathcal{L}_\tau$, τ the active tree, and $\delta_2 \in \mathcal{L}_\mu$, μ the passive tree. Furthermore, assume that $\mathcal{PATH}(\tau, \delta_1) = \mathcal{PATH}(\mu, \delta_2)$ (it is easy to remove this restriction, which is not done here), and let $\mathcal{ROUTE}(\tau, \delta_1) = ns, n \in \mathcal{NAT}, s \in \mathcal{NAT}^*$ and $\mathcal{ROUTE}(\mu, \delta_2) = mq, m \in \mathcal{NAT}, q \in \mathcal{NAT}^*$.

Most of the important information is encoded into the branch nodes. Branch nodes only arise when multiple senders contend for a single receiver. On each branch there is a synchronized communication event with the same name. The path taken from the branch node is the one that has the shortest execution time for the events that lie between the root of the branch and the synchronized communication event. Thus, the reason why a particular path is executed is that some set of events executed faster than another set.

As trees are combined, two kinds of events appear that are not present in an initial set of trees, synchronized communication events and tuples of events. Synchronized communication events have already been defined, tuples of events appear now for the first time. The additional notation needed for manipulating tuples of events is not discussed here.

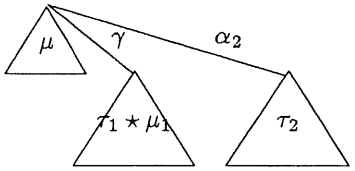
The definition of the combine operation is not quite complete. It demonstrates the case where the matching communication events contains a send and a receive pair. There is another case that occurs when the pair of matching events contains a send and a synchronized pair of events (see Figure 2). The resulting tree is a tree with a branch node at the root where one branch is the current passive tree (the tree that contains the synchronized communication event), and the other branch is the tree that results from combining the two trees in the manner shown in Figure 1. As mentioned before, combining two trees that do not contain any matching communication events results in a pseudo tree. The operators that

$\tau \star \mu$ =

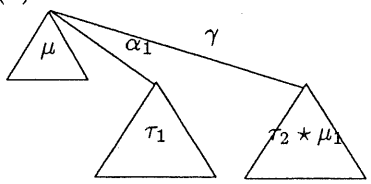


$\tau \star \mu$ =

(i) $n = 1$

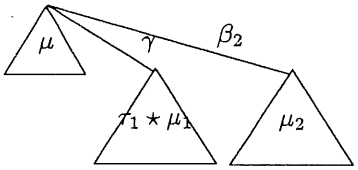


(ii) $n = 2$

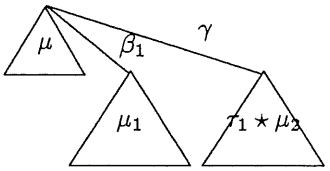


$\tau \star \mu$ =

(i) $m = 1$

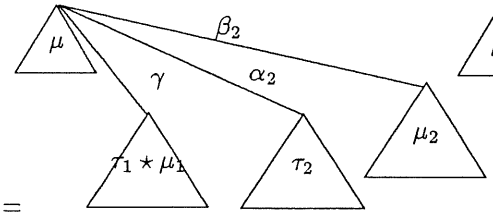


(ii) $m = 2$

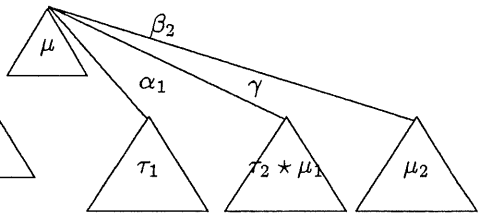


$\tau \star \mu$ =

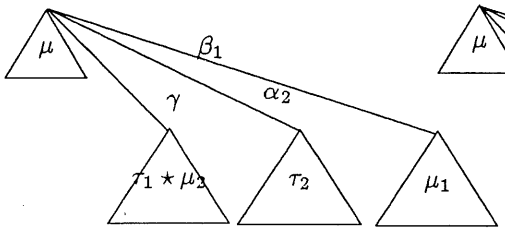
(i) $n = 1, m = 1$



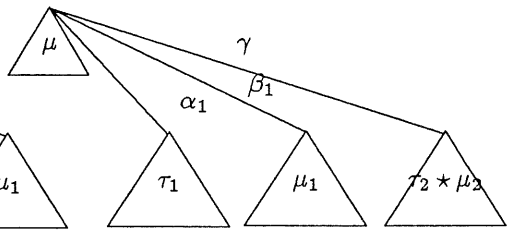
(ii) $n = 2, m = 1$



(iii) $n = 1, m = 2$



(iv) $n = 2, m = 2$



if $\alpha = a[t_1], \beta = (b_1[t_2], b_2[t_3])$
then $\gamma = (a[t_1], b_2[t_3])$

if $\alpha = a[\overrightarrow{t_1}], \beta = a[\overleftarrow{t_2}]$ then
then $\gamma = a[\overleftrightarrow{t_{1,2}}]$

$t_{1,2} = MAX(t_1, t_2)$

Figure 2

Combining Trees With Synchronized Events

define how to combine basic trees with pseudo trees and pseudo trees with pseudo trees are not described here but are denoted by $\star\star$ and $\star\star\star$ respectively. Finally, a general combine operator denoted $\circledast$ combines any two trees regardless of their respective types. However, if the set of trees to be combined is ordered (so that each pair of trees has a pair of matching communication events) then the $\star\star$, $\star\star\star$, and $\circledast$ operations are not needed. They are provided to insure that the combine operation is commutative and associative. For our purposes here they are not necessary.

Translating Programs to EDT

Translating programs to EDT is fairly straightforward. Each sequential piece of code that does not contain any communication events is assigned a unique event name. The only tricky part is assigning event names to communication events. The event names must be kept in a table along with enough information to correlate matching send and recv commands.

The translation process is now illustrated by an example. A well known example of synchronization is the dining philosphers problem. Five philosophers alternate between thinking and eating. When they want to eat they take a seat at a table that has five plates and five forks, one fork between each plate. In order to begin eating a philosopher must pick up two forks, one to his right and one to his left. If only one fork is available then he must wait for the second to become available before beginning to eat.

Figure 3 gives pseudo code for a solution to the dining philosophers problem. The version used here is taken from [HOAR85]. There is one process for each fork and one process for each philosopher. In this example only three fork and two philosopher processes are depicted. It is enough to demonstrate how programs, here written in pseudo code, are translated to event dependency trees.

```
process phil1:
  loop
    sitdown
    send[fork1, pickup]
    send[fork2, pickup]
    send[fork1, putdown]
    send[fork2, putdown]
    getup
  end
end phil1
```

```
process phil2:
  loop
    sitdown
    send[fork2, pickup]
    send[fork3, pickup]
    send[fork2, putdown]
    send[fork3, putdown]
    getup
  end
end phil2
```

```
process fork1:
  loop
    recv[pickup]
    recv[putdown]
  end
end fork1
```

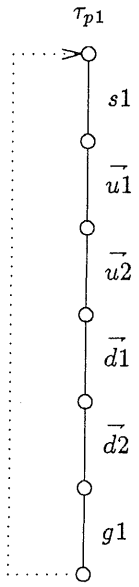
```
process fork2:
  loop
    recv[pickup]
    recv[putdown]
  end
end fork2
```

```
process fork3:
  loop
    recv[pickup]
    recv[putdown]
  end
end fork3
```

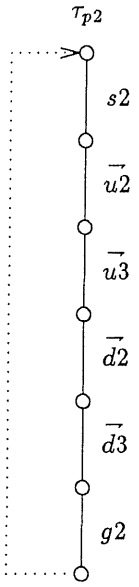
Figure 3

Dining Philosophers: Pseudo Code

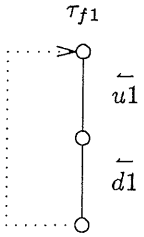
Each fork process consists of two events, first it waits for a message indicating some philosopher wishes to use the fork, then it waits for another message indicating the philosopher is finished with the fork. Each philosopher process actually represents a specific location at the table where a philosopher sits down. A philosopher process consists of six events: (1) first a philosopher sits down, (2) next he picks up the fork to his left, (3) then he picks up the fork to his right, (4) he puts down the fork to his left, (5) he puts down the fork to his right,



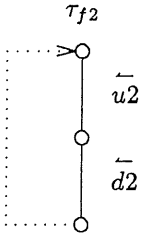
(a) Philosopher process 1



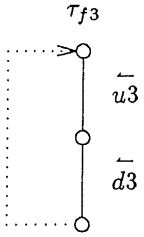
(b) Philosopher process 2



(c) Fork process 1



(d) Fork process 2



(e) Fork process 3

Figure 4

Dining Philosophers: Translated to Trees

and (6) he leaves the table. If a philosopher sends a message to pick up a fork and it isn't available, then the fork process will not be at the correct receive statement and the philosopher process will block until the fork becomes available.

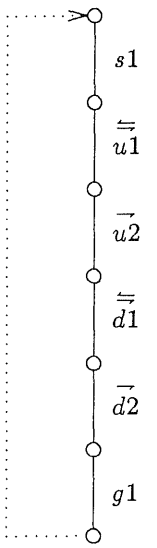
Figure 4 shows the five processes as event dependency trees. First each receive event is assigned a unique name. The only receive events occur in the fork processes. The tree representing fork process 1, called τ_{f1}, is a sequence of two events, $\overleftarrow{u1}$, which represents $recv[pickup]$, and $\overleftarrow{d1}$, which represents $recv[putdown]$.

The other two trees, τ_{f2} and τ_{f3}, which represent the processes $fork2$ and $fork3$, are the same except that the event names are unique. The send commands in the philosopher processes must be given the same event names as their respective receive commands. $send[fork1, pickup]$ is assigned the event label $\overrightarrow{u1}$. The other send commands are assigned their corresponding labels in the same manner. The actions sitdown and getup are represented as execution events, unique to each process. The five trees in Figure 4 represent the five individual processes in Figure 3.

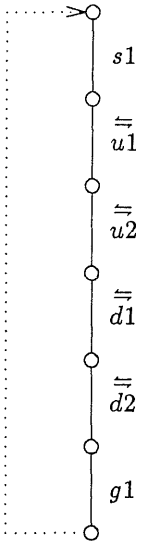
The five processes are now combined using the $\star$ operator in order to depict the system behavior. The intermediate steps and final result are shown in Figure 5. In Figure 5 part (d) the five processes are all combined. The resulting tree represents a conflict with the event $\overleftrightarrow{u2}$, which is depicted by the introduction of a branch node during the combine operation. The meaning of the branch node is: if $s1\overleftrightarrow{u2}$ occurs before $s2$, then the lefthand branch is taken. Otherwise the righthand branch is taken.

Deadlock Detection

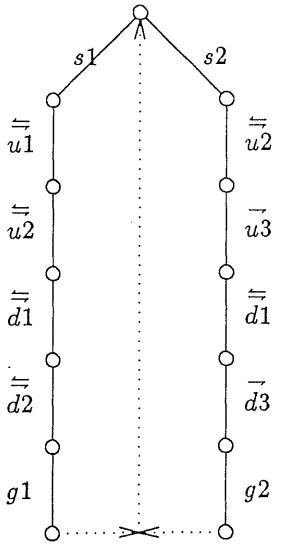
Much research has been done on deadlock detection and avoidance. Algorithms exist that are used during the execution of the system to detect deadlock and prevent it [GLIG80, KAME80, CHAN83]. Many other algorithms for deadlock detection in distributed databases have also been developed; they are similar in purpose and use to those listed above. The approach taken here is based on general use of the synchronization primitives *send* and *receive*. It is shown that deadlock due to incorrect software can be prevented before execution by using EDT.



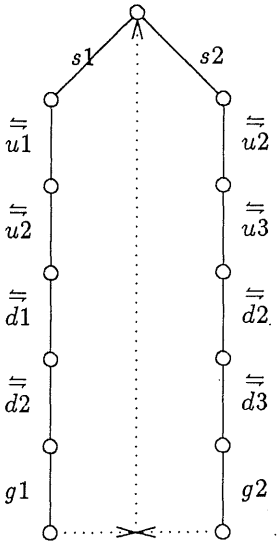
(a) $\tau_{p1} \circledast \tau_{f1}$



(b) $\tau_{p1} \circledast \tau_{f1} \circledast \tau_{f2}$



(c) $\tau_{p1} \circledast \tau_{f1} \circledast \tau_{f2} \circledast \tau_{p2}$



(d) $\tau_{p1} \circledast \tau_{f1} \circledast \tau_{f2} \circledast \tau_{p2} \circledast \tau_{f3}$

Figure 5

Dining Philosophers: Combining Trees

Use of Synchronization Primitives

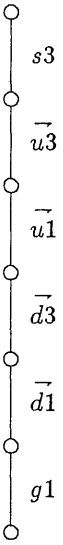
There are two sources of deadlock that can be detected using EDT, both the result of incorrect software. The first source of deadlock arises when the synchronization primitives are used incorrectly, manifested as a receiver with no sender or vice versa. This is easy to detect in the EDT representation of a piece of software. First, the processes are converted to trees (as described above), then the combine operation is repeatedly applied until all trees are combined. Any communication events in the tree that are not synchronized communication events indicate the presence of a deadlock. In Figure 5(a), if this tree represented a complete computation, then the event $\overrightarrow{u2}$ would block forever. It indicates that a process is sending a message to a receiver that doesn't exist. In the same tree $\overrightarrow{d2}$ is another unsynchronized event. To detect these problems the tree is scanned once for any receive or send events. If some are presente then a deadlock will occur.

Cyclic Dependency

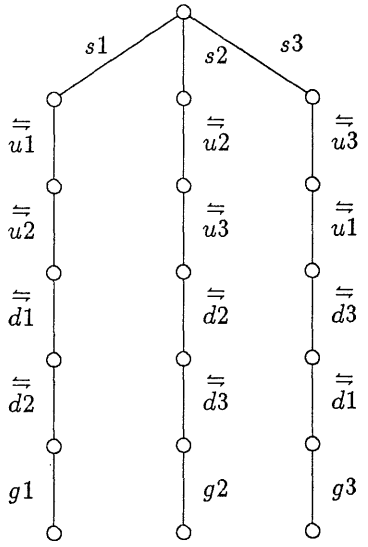
In addition to finding deadlocks that occur from incorrect use of the synchronization primitives, EDT provides some insight into how or when deadlocks might occur due to cyclic dependencies of events, even though no unsynchronized communication events are present.

Dining Philosophers

The dining philosophers problem, described above, demonstrates the second form of deadlock. Take the case of three philosophers. Another philosopher process is combined with the tree in Figure 5(d), shown in Figure 6. There are now three philosopher processes and three fork processes. Each philosopher sits down, picks up the first fork, picks up the second fork, puts down the first fork, puts down the second fork and then gets up. Sitting down and getting up are represented as execution events and the rest of the events are communication



(a) Philosopher process τ_{p3}



(d) $\tau_{p1} \circledast \tau_{f1} \circledast \tau_{f2}$

$\circledast \tau_{p2} \circledast \tau_{f3}$

$\circledast \tau_{p3}$

Figure 6

Three Dining Philosophers

events. Each fork process has two communication events, pick up fork and put down fork.

Consider the EDT $P_1 \star P_2 \star P_3 \star F_1 \star F_2 \star F_3$ in Figure 6. A deadlock problem is revealed that is due to timing. In an EDT: from a branch node only one path occurs, the path that occurs is the one whose events from the root of the tree to the common communication event occur first. Thus if $s_1 \overleftrightarrows u_1$ occurs before s_2 completes then the first branch will be taken. However, consider the case where $s_2 \overleftrightarrows u_2$ occurs before $s_1 \overleftrightarrows u_1$ and $s_1 \overleftrightarrows u_1$ occurs before $s_3 \overleftrightarrows u_3$. This is a deadlock.

In reality an EDT like the one just discussed is never allowed (recall the restriction about combining trees with multiple pairs of matching communication events). In terms of EDT's, the correct structure is to have each path trying to synchronize on the same event. In Figure 6 the three combinations of pairs of processes are trying to synchronize on three different events. The first and third branches are contending for $\overleftrightarrows u_1$, the first and second branches are contending for $\overleftrightarrows u_2$, and the second and third branches are contending for $\overleftrightarrows u_3$. If any one branch is removed, then the remaining two branches are only contending for one synchronised event, which is a correct structure.

The original rule about combining trees states that two trees with multiple pairs of matching communication events can not be combined if branch nodes occur between the pair of events in one tree, and the events occur within a chain in the other tree. Therefore the tree in Figure 6 would never have been produced in the first place. Any two philosopher processes can be combined with the tree fork processes as in Figure 5 but the restriction is violated when the remaining philosopher process is introduced.

Now consider a solution to the deadlock problem. Another process is introduced that only allows two philosophers to sitdown at any given time. This corresponds quite closely to the structure revealed in the EDT representation.

```
        i := 0
here:
        recv [sitdown]
        i := i+1
        if i = 1
            then recv [m]
                 if m = sitdown
                     then i := i+1
                     else i := i-1
        elsif i = 2
            then recv [getup]
                 i := i-1
        go to here


m is a message that can be either sitdown or getup
```

Figure 7

A Solution to Cyclic Dependency

It is not possible, however, to represent the solution directly because it contains control structures which can not be modelled by EDT at this time. Figure 7 contains code for the solution. In essence it requires all philosopher processes to first synchronize with a new process on the event *sitdown*. Then a count of the number of philosophers eating is kept. The message m can be either *sitdown* or *getup*. Never are more than two philosophers allowed to sit down before one must get up. The *getup* event also becomes a synchronized event. In terms of the EDT representation (if it could be represented) this requires every branch to synchronize on just one event, $\overleftrightarrow{sitdown}$ and then any two of the three paths will further synchronize on a fork event.

Summary

EDT is a formal model of distributed or communicating systems that predicts how CSP–type processes will interact. Although it appears that EDT is a model of software, assumptions about how the system impacts the execution of the software is a crucial aspect of the model, the primary assumption being that events take time that could differ from execution to execution.

Next it was shown how Event Dependency Trees can be used to produce deadlock-free concurrent software. First a program or pseudo code is translated into an EDT representation. At this stage there is one tree for each process in the software. Then the combine operation is applied until all trees are glued together. During the combine operation, deadlock resulting from a cyclic dependency of events is revealed. It is detected from structural properties of the two trees being combined. If no cyclic dependencies are detected then a representation of the system is the result.

At this stage more deadlock detection is performed to discover if there exist any unsynchronized communication events. If there are the software is corrected, translated to trees, and combined. At this point if no deadlocks due to cyclic dependencies or incorrect matching up of send and receive commands are present. The model supplies potentially important information for the design and construction of concurrent software systems.

References

[Broo84] Brookes S.D., Hoare C.A.R., and Roscoe A.W. A Theory of Communicating Sequential Processes. *Journal of the ACM 31*, 3 (July, 1984), 560–599.

[Chan83] Chandy K.M., and Misra J. Distributed Deadlock Detection. *ACM Transactions on Computer Systems 1*, 2 (May, 1983), 144–156.

[Glig80] Gligor V.D., and Shattuck S.H. On Deadlock Detection in Distributed Systems. *IEEE Transactions on Software Engineering 6*, 5 (September, 1980), 435–440.

[Henn85b] Hennessy M. Acceptance Trees. *CACM 32*, 4 (October, 1985), 896–928.

[Hoar85] Hoare C.A.R. *Communicating Sequential Processes*, Prentice–Hall, 1985.

[Kame80] Kameda T. Testing Deadlock–Freedom of Computer Systems. *Journal of the ACM 27*, 2 (April, 1980), 270–280.

[Miln80] Milner R. *A Calculus of Communicating Systems*, Goos G., and Hartmanis J., Ed., Springer–Verlag, Berlin, 1980.

[Wins84] Winskel G. Synchronization Trees. *Theoretical Computer Science 34* (1984), 33–82.