IEEE TSE 16, 11, November 1990

The Evolving Philosophers Problem: Dynamic Change Management.

Jeff Kramer and Jeff Magee

ABSTRACT

One of the major challenges in the provision of distributed systems is the accomodation of *evolutionary* change. This may involve modifications or extensions to the system which were not envisaged at design time. Furthermore, in many application domains there is a requirement that the system accomodate such change *dynamically*, without stopping or disturbing the operation of those parts of the system unaffected by the change. Since the description of software structure (components and interconnections) provides a clear means for both system comprehension and construction, it seems appropriate that changes should also be specified as structural change, in terms of component creation/deletion and connection/disconnection. These changes are then applied to the operational system itself to produce the modified system.

This paper presents a model for dynamic change management which separates structural concerns from component application concerns. This separation of concerns permits the formulation of general structural rules for change at the configuration level without the need to consider application state, and the specification of application component actions without prior knowledge of the actual structural changes which may be introduced. In addition, the changes can be applied in such a way as to leave the modified system in a consistent state, and cause no disturbance to the unaffected part of the operational system. The model is applied to an example problem, "evolving philosophers". The principles described in this model have been implemented and tested in the Conic environment for distributed systems.

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate,
London SW7 2BZ, UK.

1. INTRODUCTION

Distributed computing systems are in widespread use in commercial, industrial and research establishments. One of the major difficulties in the development and maintenance of such systems is that of systems management, particularly with respect to the management of change. Distributed systems need to evolve as human needs change, technology changes and the application environment changes. It has been argued [18] that the introduction of the computing system is itself a stimulus for change. These changes may require modification of a function already provided by the system, or extension by the introduction of new functions. In general, these changes, termed *evolutionary*, are difficult to accomodate as they cannot be predicted at the time the system is designed. Consequently, we would like systems to be sufficiently flexible to permit arbitrary, incremental change. In addition, we believe that systems should be capable of supporting such change *dynamically*, without interrupting the processing of those parts of the system which are not directly affected. Hence, it should be possible to direct changes at the operational system itself.

Distributed systems potentially offer a flexible environment for dynamic modification and extension [4,15]. The underlying support mechanisms for change (software component creation, binding and deletion) are readily available. However, there has been little suggestion as to how such dynamic change should be specified, managed and controlled. This paper describes an approach based on a separation of concerns: functional concerns of the application processing components which can be provided with a general capability for change independent of what structural changes are introduced, and structural configuration concerns for specifying structural change without the need to consider application state.

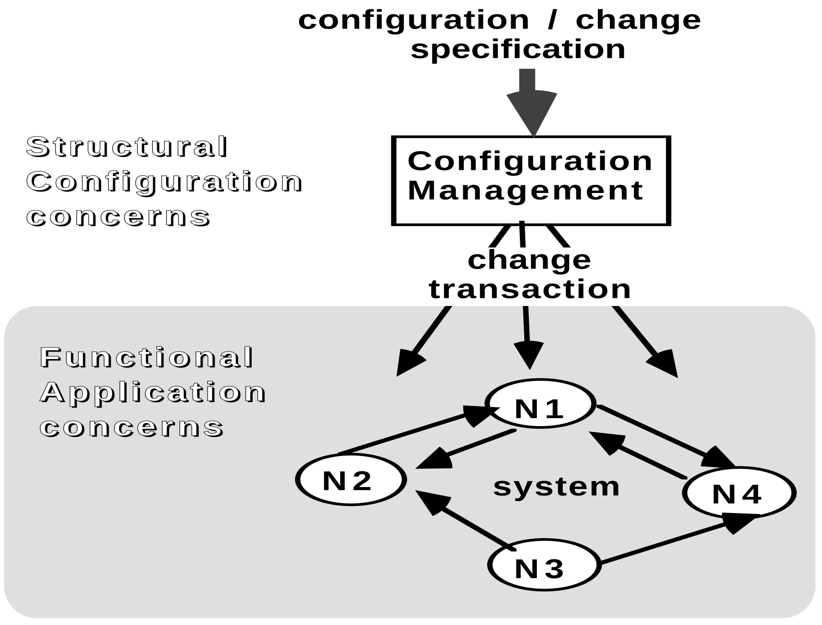

configuration / change specification
Structural Configuration concerns
Configuration Management
change transaction
Functional Application concerns
N1
N2
system
N4
N3

Figure 1.1 System Configuration and Change Management

Recent work on distributed software specification and construction confirms the benefits of separating the software component programming concerns from those of system configuration

[3,15,17]. A separate configuration specification is useful both as a description of the system structure and to generate the actual system. As for software construction, we believe that change is well handled at the configuration level in terms of software components and their interconnections [10,14]. Figure 1.1 identifies the central role of configuration management which is required to interface between the functional view of application programming and the structural view of system configuration. Changes are specified declaratively in terms of system structure only. The system itself is modified by the application of procedural change transactions, which include ordered sets of structural and control actions. These change transactions are derived by management from the change specifications. Hence, the system evolves incrementally by the application of change specifications as shown in figure 1.2. Alternatively, a change specification could be derived by comparing the desired configuration specification with that of the current system.

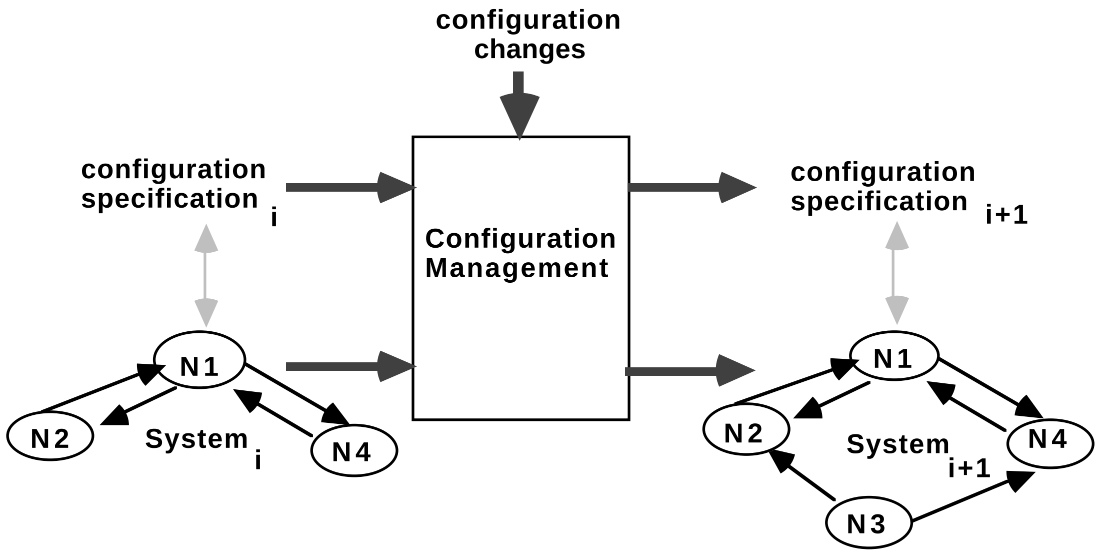


Figure 1.2 - Evolution of a system by incremental changes.

In addition to the means for specifying and performing change, it is also necessary to provide facilities for controlling change such that application consistency is preserved both while the change is applied and subsequent to the change. This is the role of change management.

What exactly are the required characteristics of a configuration management system for managing dynamic change? Section 2 identifies the objectives of change management, and defines more precisely the distributed system environment in which it is expected to operate. The form of interaction between management and the application nodes (software components) is presented in Section 3, defining the management view of node states and the desirable properties which this provides. This leads to the development of a management protocol in section 4 which describes how change management is applied, illustrating the process with some simple examples. It also briefly discusses the required application contribution to the change process. This is followed in section 5 by a detailed example, the Evolving Philosophers. Section 6 extends the work to cover systems with more interdependence, and illustrates the principles for the evolving philosophers problem. The concluding section of the paper discusses other approaches to dynamic change management which range from pragmatic support for procedure replacement to the more formal transformational approach. The conclusions also examine the adequacy of our approach and discuss its implications.

2. DYNAMIC CHANGE MANAGEMENT

This section identifies the objectives of change management, and defines more precisely the distributed system environment in which it is expected to operate. These objectives represent an approach which clearly separates application specific functions from structural configuration functions. As mentioned, the intention is to provide an application independent configuration management facility.

Objectives:

Changes should be specified in terms of the system structure.

Systems, and in particular distributed systems, are constructed in a modular way consisting of a configuration of software nodes. We propose that system evolution at the level of programming in the small [6] is at too low a level, being both too detailed and impractical due to the tight coupling of program elements. Instead, change should be supported at a component node level (or levels) where the possibility exists for understanding the effects of change and where the inter-node coupling is such that change is both possible and pragmatic.

Change specifications should be declarative.

By this we mean that it should be the responsibility of the configuration management system, not the user, to determine the specific ordering of actual change operations applied to the system. This clearly separates the change required (which is application specific) from how it is to be executed. Configuration management can exploit parallelism for implementing changes. Declarative specifications leave such decisions to the implementation mechanisms.

Change specifications should be independent of the algorithms, protocols and states of the application.

In order to provide generic configuration management, the configuration management system which carries out the changes should be independent of the application. Since changes are to be specified in terms of the configuration structure, the paper will demonstrate that dependence on aspects such as application state can be abstracted to a general requirement for application quiescence, and dependence on application algorithm to a need for component connection initialisation/finalisation actions.

Changes should leave the system in a consistent state.

Informally, a consistent application state is one from which the system can continue processing normally rather than progressing towards an error state. It is usually expressed in terms of some global system invariant. A system is viewed as moving from one consistent state to the next. In fact, application transactions modify the state of the application, and, while in progress, have transient state distributed in the system. While transactions are in progress the internal states of nodes may be mutually inconsistent. In order to avoid the loss of application transactions and achieve a consistent state after change, a consistent application state is required in the affected part of the system before the change .

Changes should minimise the disruption to the application system.

It should not be necessary to stop the whole of a running application system to modify part of it. The management system should, from the change specification, be able to determine a minimal set of nodes which are affected by the change. The rest of the system should be able to continue its execution normally.

These objectives have a number of consequences for both the application and the configuration management system. In particular, the management system must give the affected part of a system the opportunity to reach a consistent state before a change is performed. The management system does not force changes but waits for the application to reach a consistent state. This consistent state requires that there is no communication in progress between the affected nodes nor with their environment. Each node is said to be in a **quiescent** state. Further, the nodes must remain quiescent while the change is executed. This gives newly created nodes the opportunity to be initialised in a state which is consistent with the rest of the system and nodes which are being removed the opportunity to leave the system in a consistent state. Later sections will describe how the affected nodes are identified and controlled by management, and discuss the application level responsibilities.

Distributed System Model.

In order to provide a sound basis for a discussion on change management, we first describe the environment and the assumptions made. We also briefly define the terms used.

• **System** - A system is assumed to consist of a set of processing nodes with directed connections indicating the communication paths between the nodes.

• **Node** - A node is a processing entity which can initiate and service transactions.

• **Connection** -A connection is a directed communication path from the initiator of the communication exchange to the recipient (figure 2.1)

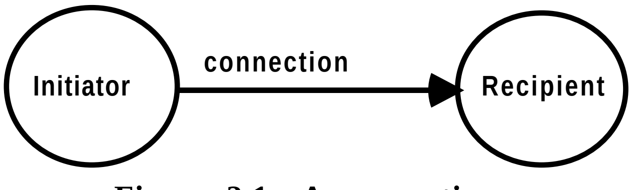


Figure 2.1 A connection.

A system may thus be represented as a directed graph (Figure 1.1). The edges in the figure indicate that node N1 may initiate and receive transactions with both of N2 and N4. N2 may only initiate transactions with N1, but may receive transactions from nodes N1 and N3.

• **Transaction** - A transaction is an exchange of information between *two and only two* nodes, initiated by one of the nodes. Transactions are the means by which the state of a node is affected by other connected nodes in the system. Transactions consist of a sequence of one or more message exchanges between the two connected nodes. It is assumed that transactions complete in bounded time and that the initiator of a transaction is aware of its completion.

Figure 2.2 illustrates valid examples of transactions. In practice they may consist of a remote procedure call (rpc) or request-reply message exchange as in (a), or some sequence or combination of rpc's or message passing as shown in (b). The only requirement is that one of the two parties is identifiable as the initiator of the transaction and is informed of the completion of the transaction. Completion of transactions at the initiator is required to ensure correct termination of the management protocol described later. We assume only **independent transactions**, where completion of a transaction does not depend on any other (possibly nested) transactions with other nodes. Section 6 discusses the implication of relaxing this restriction to permit **dependent** transactions, where completion is dependent on **consequent** transactions with other nodes.

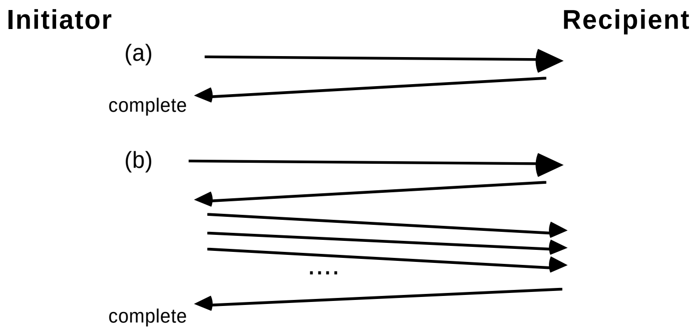


Figure 2.2 - Examples of (two-party) Transactions

• **Change** - A change is described in terms of modifications to the structure (configuration) of the application system. Changes take the form of node creation and deletion, and connection establishment and removal. Changes are effected by a Configuration Manager. Previous work [14,19] has identified a set of management primitives for both specifying and modifying the structure of systems. In abstract form these are:

create N:T [**at** L]

Create node N of type T, optionally specify at physical location L.
The name N must be unique within the system.
(For simplicity, T is omitted in later examples where the type is obvious).

remove N

Remove node N

link N1 **to** N2

Create a connection from node N1 to node N2.
(For simplicity, we omit the detail of multiple connections between nodes since this does not alter the algorithms presented in the following.)

unlink N1 **from** N2

Remove a connection between node N1 and node N2.

• **Consistency** - This is determined by the relationship between node application states and is usually described by some global invariant (constraint) which must be preserved. For local consistency of a node, it is necessary that there are no partially complete transactions at the node.

The next section describes the interface between management and the application nodes and refines the notion of quiescence.

3. APPLICATION - MANAGEMENT INTERACTION

When performing configuration changes it is important that application information is not lost and that the application is left in a consistent state. To do this, the management system should have an interface with the application which allows it to direct the application towards an appropriate state for reconfiguration. Further, the management system must be able to confirm that the application has reached this state. The interface between application and management system must be a generic one which makes management independent of the particular application. To meet this objective, application state is abstracted into a set of configuration management states for each node. This set of states provides sufficient information about application state to allow the management system to perform changes which leave the application in a consistent state. These configuration states and the transitions between them are outlined below.

Node Configuration States and Transitions

The state transition diagram of Figure 3.1 specifies the possible states for an application node from the configuration viewpoint (cf. process states from an operating system viewpoint). The interaction between configuration management and the node are indicated as transitions. These transitions are instigated by configuration management (cf. "control actions" [8]) and should be distinguished from the normal application level transactions ("basic communications").

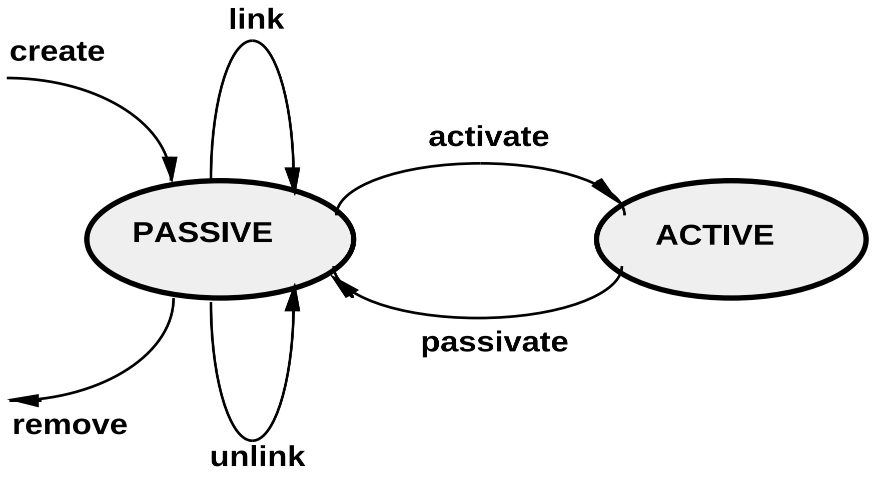
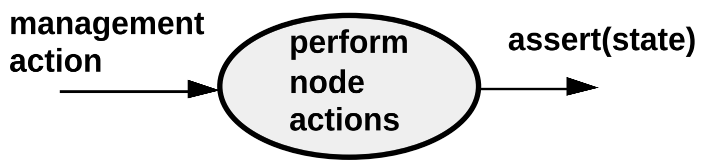


Figure 3.1 - Node State Transitions

Transitions

Each transition depicted in Figure 3.1 represents the management action which initiates the transition. Node application actions are required to reach the destination management state (Figure 3.2).



.

Figure 3.2 - Node Transitions

The temporary state involved in each transition gives a node the opportunity to perform the initialisation and finalisation actions necessary to preserve application consistency when it is **created/removed** and when connections are **linked/unlinked**. The application indicates that these actions are complete by asserting the destination state of the transition to the change management system. **Activate** and **passivate** (figure 3.1) are the transitions between the active and passive states described below.

States

- A node in the **active** state can initiate, accept and service transactions.

- A node in the **passive** state must continue to accept and service transactions, but
  (i) it is not currently engaged in a transaction that it intiated, and
  (ii) it will not initiate new transactions.

The particular state identified as necessary for reconfiguration is the passive state. A node in the passive state must continue to *accept* and *service* transactions while it is in the passive state, but it must *not initiate* any new transactions as a result of accepting or servicing transactions. This passive state is so defined as to permit connected nodes to progress towards a passive state by completing outstanding transactions. In addition it contributes to system consistency by completing transactions. However, the passive state is not sufficient for reconfiguration as it may still be processing transactions initiated by other nodes.

Node Quiescence

For consistency during change we require, a stronger property, viz. that the node is not within a transaction and will neither receive nor initiate any new transactions. This property is called **quiescence** of a node and is that state in which the node is both passive and has no outstanding transactions which it must accept and service. Such a state depends not only on the node itself, but on the connected nodes.

Consequently, a node is **quiescent** if:

(i) it is not currently engaged in a transaction that it initiated, } passive
(ii) it will not initiate new transactions, } properties
(iii) it is not currently engaged in servicing a transaction, and
(iv) no transactions have been or will be initated by other nodes which
    require service from this node.

In the quiescent configuration state, the application state of a node is both **consistent** and **frozen.** It is consistent in that the application state does not contain the results of partially completed transactions, and is frozen in that the application state will not change as a result of new transactions. Quiescence is significant for dynamic configuration changes since, in cases such as unlinking, it permits a node to make decisions based on a stable and consistent state regarding the particular actions it should take before it is unlinked. For instance, the node may pass a consistent uptodate version of its application state to its environment before it is unlinked.

Our notion of quiescence is loosely based on earlier work [13] which specified node behaviour using a "quiescent invariant": the stable, steady state properties of a node characterised by a local invariant preserved by the node. Quiescence is also related to that defined by Misra [20], except that the discussion there focuses on traces and termination, whereas we focus on node state and consistency. Misra defines a node as quiescent if it *may not* produce further output. Our notion is stronger in the sense that the trace produced by a quiescent node *will not* be extended by any further output.

Resultant Properties for Systems using Independent Transactions

Given that the passive and quiescent states are desirable node management states, we now show how they can be achieved for systems constructed from **independent** transactions. From section 2, an independent transaction is a two party transaction whose completion does not depend on any other transaction. In these systems, a transaction serviced by a node may cause that node to initiate transactions to other nodes, however, completion of the service may **not** depend on completion of any transaction which the node may initiate. Section 6 discusses the extensions required for **dependent** transactions.

The following propositions and justifications demonstrate the reachability of the passive state, the relationship between passive and quiescent states and the reachability of the quiescent state.

PROPOSITION 1: (Reachability of the Passive state)

In independent systems, a node can move from the active to the passive state in bounded time, irrespective of the configuration state (active or passive) of the nodes to which it is connected.

JUSTIFICATION 1:

To be passive, the node must satisfy two conditions:
(i) it is not currently engaged in a transaction that it initiated:

A node will complete in bounded time any transaction which it initiated since transactions complete in bounded time and completion is independent of the completion of transactions at other nodes. Transactions complete in bounded time even if the recipient node is in the passive state since passive nodes accept and service transactions.
(ii) it will not initiate new transactions.

This property can be immediately satisfied by the application.

For systems using independent transactions, we define the passive set **PS** of a node **Q**, denoted **PS(Q)**, to consist of:

1) the node **Q**
2) all nodes which can directly initiate transactions on **Q**, i.e. all nodes with connection arcs directed towards **Q.**

PROPOSITION 2: (Passive requirements for the Quiescent state)

In systems using independent transactions, Q is quiescent if all nodes in PS (Q) are in the passive state.

JUSTIFICATION 2:

A node is quiescent if:
(i) it is not currently engaged in a transaction that it initiated,
(ii) it will not initiate new transactions,
(iii) it is not currently engaged in servicing a transaction, and
(iv) no transactions have been or will be initated by other nodes which require service from this node.

Conditions (i) and (ii) follow from the passive state of Q ie. Q is in PS(Q).
Conditions (iii) and (iv) follow from the passive states of the nodes in PS(Q), i.e. if all nodes which can initiate transactions on Q are also passive, then all transactions involving Q are complete and no new ones will be initiated.
Hence Q is in a quiescent state.

PROPOSITION 3: (Reachability of the Quiescent state)

In systems using independent transactions, a node Q can move from the active to the quiescent state in bounded time if all the nodes in PS(Q) are directed to move into the passive state.

JUSTIFICATION 3:

Since all nodes will achieve the passive state in bounded time (Proposition 1), and the passive state of all nodes in PS(Q) imply quiescence of Q (Proposition 2), then Q will achieve the quiescent state in bounded time.

This section has defined an interface through which a configuration manager communicates with and controls an application node. Communication between configuration management and the node is synchronous in the sense that a management action is always confirmed by the node. (For pragmatic reasons, it may be necessary to support the forced removal of "rogue" nodes which do not obey or react correctly to configuration commands. These can be added as **remove** transitions from the active state.) Note that the node configuration state is the only way that configuration management can affect the application state. The passive state has been carefully defined to be readily achievable by a node by completion of any transactions which it initiated. Since transactions complete in bounded time, the passive state can be achieved in bounded time. Similarly, since the passive state permits servicing of transactions initiated by connected nodes, they too will be permitted to progress to a passive state. However, for the configuration manager to achieve quiescence of some target node, it is necessary to make the target node passive and also to create a region of passive nodes (the passive set) around it. This will achieve a stable situation where there are no incomplete or active transactions. This together with the abstraction of application state into configuration management states forms the basis of the change protocol outlined in the next section.

4. CHANGE MANAGEMENT PROTOCOL

MANAGEMENT VIEW

In this section we outline a change protocol for systems constructed from independent transactions. This protocol meets the change management objectives of section 2. In particular, the objective of a declarative, as opposed to imperative, change specification means that changes are specified using only structural actions **create**, **remove**, **link** and **unlink** (see section 2). The **activate** and **passivate** actions on nodes are essentially an implementation device which should not be visible to a user. The following outlines a change protocol in which the change *transactions*, including activate and passivate actions and the ordering of execution, can be automatically derived from the change *specification* (see figure 1.1).

Change Rules

The change protocol involves establishing a region of quiescence, specified as the set of nodes required to be passive, where the change is to occur. As mentioned in section 2, changes involve node creation and deletion, and connection establishment and removal. We now examine each of the possible changes in turn and present the rules for contributing nodes to the passive set.

(i) Node deletion - remove

Rule: The precondition for removing a node N is that it is **isolated**. By isolated, we mean that it has no connections directed to it from other nodes or from it to other nodes.

Justification: An isolated node cannot affect the system and so can be independently removed.

(ii) Connection - link and unlink

Rule: The precondition for either linking or unlinking is that the node N from which the connection is directed must be in the quiescent state. (From Proposition 2, we know that this can be achieved by requiring that all nodes in PS(N) are in the passive state.)

Justification: Quiescence of the initiator node ensures that its state is consistent and frozen with respect to that connection, thereby enabling connection initialisation/finalisation to occur in a stable environment.

(iii) Node Creation - create

Rule: The precondition is trivially true.

Justification: When a node is created it is initially isolated and consequently must be in the quiescent state since it can neither respond to nor initiate transactions on other nodes.

Change Transactions

A change transaction consists of a set of partially ordered configuration actions (or commands), which is derived from the structural change so as to satisfy the pre-conditions outlined above. One possible algorithm for deriving change transactions is outlined below.

Step 1) Determine the the set of connections **CS** which must be unlinked to isolate nodes to be removed (to satisfy **(i)**). From this, together with the set of connections **LS** directly specified in link or unlink directives, determine the set of nodes **QS** (**Quiescent Set**) which must be made quiescent to satisfy **(ii)** above.

ie. CS = {connections c | c is a connection to/from a node to be removed}
LS = {connections l | l is a connection in a link/unlink directive}
QS = { nodes n | n is the initiator node on a connection in (CS ∪ LS)}

Step 2) Form the change passive set **CPS** as the union of passive sets PS of each node in QS.

ie. CPS = ∪ PS(i) **forall** i in QS

Step 3) Perform the configuration actions in the following order:

```
passivate   <all nodes in the change passive set CPS >
unlink
remove
create        (could be performed at any time before link)
link
activate    <CPS - removed nodes + created nodes>
```

It should be noted that if the change management system permits multiple change transactions to be performed in parallel then the set of nodes which must be locked for a change is a superset of the change passive set. In detail, the lock set is:

LockSet = CPS ∪ { nodes n | n is a *recipient* node on a connection in (CS ∪ LS)}

The lock set includes nodes to which connections are directed so that a change transaction does not attempt to make a connection to a node which has been deleted by another concurrent transaction.

A Simple Example

In order to illustrate the management view of the change protocol, we briefly describe some possible changes for a simple client-server system. The system graph for the client-server system is shown in figure 4.1.

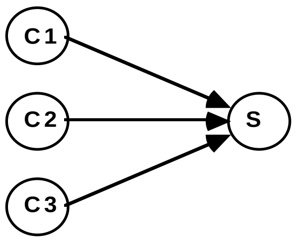


Figure 4.1 A simple client-server system

(i) Adding a client : -	Change Passive Set = {}
(ii) Unlinking or Removing a client C1 :-	Change Passive Set = {C1}
(iii) Removing the server S :-	Change Passive Set = {C1,C2,C3,S}

S is passive when all client transactions have completed. In order to replace the server S with a modified server S', the following change specification is provided:

```
change specification:
        remove S            <and unlink dangling connections>
        create S'
        link C1 to S', C2 to S', C3 to S'
```

and the derived change transaction is as follows:

```
change transaction:
        passivate C1, C2, C3, S  &  create S'
        unlink C1 from S, C2 from S, C3 from S   <unlink dangling connections>
        remove S
        link C1 to S', C2 to S', C3 to S'
        activate C1, C2, C3, S'
```

APPLICATION CONTRIBUTION

The description so far has concentrated on the management view of change. System consistency is an application dependent notion and in general requires nodes to contribute to its preservation. One of the main contributions that an application node must make is preservation of the **passive** state i.e. the application must not initiate any new transactions, but must be prepared to service transactions from other nodes. The change from active to passive state is implemented in a node by converting the general description of the passive state into an invariant constraint in application terms (ie. referring to local node variables) that must be preserved by the node. When in the passive state, the node confirms this by responding **assert(passive)** (Figure 3.2) to management.

Furthermore, in order for a newly connected application node to preserve consistency, it must be given the opportunity to initialise itself so as to be consistent with its new environment. Similarly a node which is about to be disconnected must be given the opportunity to clean up in order to leave its environment in some consistent state. These opportunities are provided for in the management protocol by the **link** and **unlink** transitions (figure 3.1) where the node can, if necessary, execute such actions as necessary. These actions may include communication with other nodes. Again, completion of these actions is confirmed by an **assert(passive)** response to management.

Obviously the actual actions which need to be executed at these times are application dependent. However, these are simplified by the fact that, by the change rules described above, there is no transient information in the node. In general, these actions may include the initiation of queries on connected nodes. The complexity depends on the complexity of the application and the autonomy of the node. This confirms our intuition that, if a system is designed such that its constituent nodes are tightly coupled and interdependent, then the connection initialisation/finalisation actions are likely to be correspondingly complex.

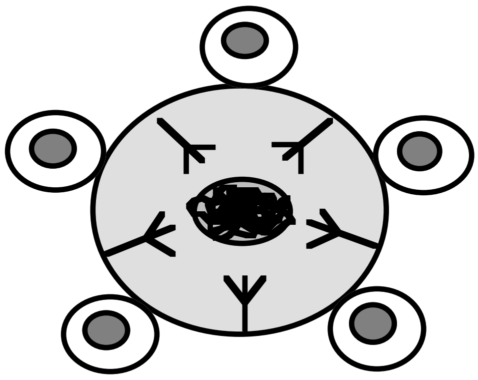
The application contribution is illustrated in the detailed example in the next section.

5. EVOLVING PHILOSOPHERS

To illustrate the management scheme developed in the preceding sections, it is applied to the Dining Philosopher's (Diners) problem [7]. Philosophers are arranged in a ring with neighbouring philosophers sharing a fork. A philosopher is either thinking, hungry or eating. To move from the hungry to the eating state a philosopher must acquire both his lefthand and righthand fork. The solution presented below is a modification to the fully distributed diners solution due to Chandy and Misra[5] to permit dynamic change. First, we outline Chandy and Misra's solution for a static number of philosophers and then describe the modifications necessary to permit arbitrary changes to a dining philosopher system such as the addition/deletion (birth/death) of philosophers and the merging/splitting of communities of philosophers. Coping with these dynamic changes is the evolving philosophers problem. Based on the change model, a solution is described below. This solution has been implemented and tested in the Conic environment for distributed programming [14,15,19].

Chandy and Misra's Hygenic solution to the Diners Problem

Each philosopher P_i is implemented as a process which communicates with its left- and righthand neighbours by asynchronous message passing. The system structure is depicted in Figure 5.1.



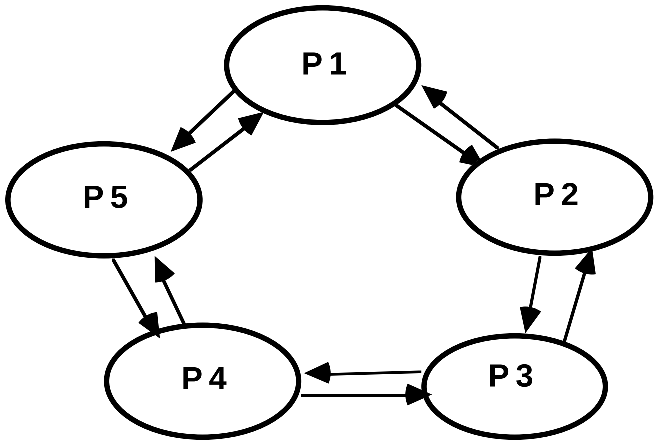
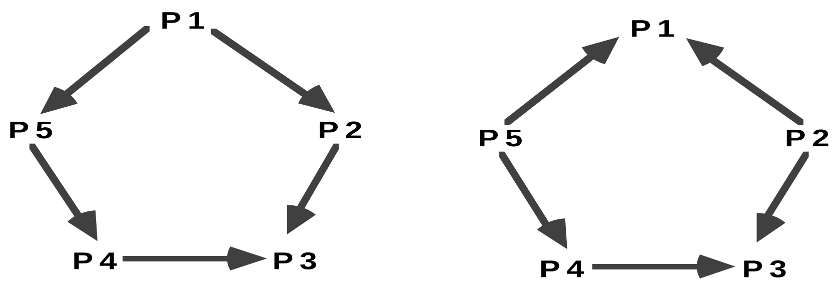


Figure 5.1 -Schematic and Structure of the Diners System

Chandy and Misra describe their solution informally as follows: " *A fork is either clean or dirty. A fork being used to eat with is dirty and remains dirty until it is cleaned. A clean fork remains clean until it is used for eating. A philosopher cleans a fork when mailing it (he is hygenic). An eating philosopher does not satisfy requests for forks until he has finished eating.*" When not eating, philosophers defer requests for forks that are clean and satisfy requests for forks that are dirty. This solution can be considered to implement a precedence graph such that an edge directed from a node **u** to **v** indicates that **u** has precedence over **v**.(Figure 5.2.)



(a) P1 hungry **(b) P1 eating**

Figure 5.2 - Precedence graph

In the diners solution a philosopher node **u** has precedence over its neighbour **v** if and only if (1) **u** holds the fork and it is clean, or (2) **v** holds the fork and it is dirty, or (3) the fork is in transit from **v** to **u**. Chandy and Misra showed that if initially all forks are dirty and located at philosophers such that the precedence graph is acyclic it will remain acyclic since (1) the direction of an edge (from **u** to **v**) can only change when **u** starts eating and (2) both edges on a philosopher are simultaneously directed towards him when he starts eating. Chandy and Misra prove that since immediately on finishing eating a philosopher yields precedence to his neighbours, all hungry philosophers will commence eating in finite time i.e. no philosopher remains hungry forever.

More precisely the algorithm is described as follows:

messages:

forktoken_f:	*passes fork f to neighbour which shares f (f can take the value **left** or **right**)*
reqtoken_f:	*passes request token for fork f to neighbour*

boolean variables:

fork(f):	*philosopher holds fork **f***
reqf(f):	*philosopher holds the request token for fork **f***
dirty(f):	*fork **f** is at philosopher and is dirty*
thinking/hungry/eating:	*state of philosopher*

Initialisation:

1) all forks are dirty
2) forks distributed among philosophers such that.the precedence graph is acyclic.
3) if **u** and **v** are neighbours then either **u** holds the fork and **v** the request token or vice versa.

The algorithm for each philosopher is described as a set of rules **guard=>action** which form a single guarded command.

```
(R1) Requesting a fork f:
        hungry,reqf(f),~fork(f) => send($\text{reqtoken}_f$); reqf(f):=false
(R2) Releasing a fork f:
        ~eating,reqf(f),dirty(f)=>  send($\text{forktoken}_f$); dirty(f):=false; fork(f):=false
(R3) Receiving a request token for f:
        receive($\text{reqtoken}_f$)=>  reqf(f):=true
(R4) Receiving a fork token for  f:
        receive($\text{forktoken}_f$)=> fork(f):=true  {~dirty(f)}
(R5) Philosopher hungry to eating transition:
        hungry,fork(left),fork(right)=>
```

```
                    eating:=true; hungry:=false; dirty(left):=true; dirty(right):=true;
  (R6) Philosopher eating to thinking transition:
              eating,eating time expired=> thinking:=true; eating:=false
  (R7) Philosopher thinking to hungry transition
              thinking,thinking time expired => hungry:=true; thinking:=false
```

AN EVOLVING COMMUNITY OF DINING PHILOSOPHERS

In the following we consider the creation of a ring (community) of philosophers, addition of a new philosopher (birth) and deletion of an existing philosopher (death). Major changes in the community are performed as the merging/splitting of communities of philosophers.

Application Contribution for Dynamic Change

To permit philosopher nodes to be subject to change we must extend the above algorithm to support the management/application interface described in section 3. In particular, the algorithm must support the passive state and provide initialisation (finalisation) actions when a philospher is linked (unlinked) to (from) another philosopher. The consistency requirements in the system are:

(i) that a fork is always shared between two adjacent, connected philosophers, and

(ii) that the precedence graph remains acyclic.

The base case of a single philosopher node is taken care of by connecting it to itself, thereby permitting it to possess two forks.

Philosopher Passive State

From the definition of passive (Section 3) a philosopher is in the passive state if firstly, it is not currently engaged in a transaction which it has initiated (i.e it has not sent a *reqtoken* which has not yet been answered by a *forktoken*) and secondly, it will not initiate new transactions ie. it is not hungry and it will not become hungry.

active:	*true when this philosopher is in the* ***active*** *management state*
passive:	*true when this philosopher is in the* ***passive*** *state.*
activate:	*management request to make philosopher* ***active*** *.*
passivate:	*management request to make philosopher* ***passive*** *.*

```
(R7)* Philosopher thinking to hungry transition:
          active,thinking,thinking time expired =>
                hungry:=true; thinking:=false
(R8)  passive to active  transition:
          activate => assert( active )
(R9)  active to passive management transition:
          ~hungry, passivate => assert(passive )
```

The modified rule (R7)* ensures that a philosopher can only move into the hungry state when it is **active**. Rule (R9) ensures that a philosopher can only become passive when it is not hungry. Since a philosopher can only initiate and be engaged in transactions when it is in the hungry state, the above rules satisfy the management requirements for active and passive states. Note that, when neighbouring philosophers are both passive, neither is hungry. In this case, the shared fork will be dirty and the precedence edge will be directed towards the holder of the dirty fork.

Philosopher Link and Unlink actions

The following rules deal with the actions required to unlink and link a connection between philosophers.

messages:

new_forktoken$_f$:	*passes new fork f to neighbour which shares f for connection initialisation*
new_reqtoken$_f$:	*passes new request token for fork f to neighbour*
link(p$_f$):	*management request to initialise a connection to philosopher which shares fork f with this philosopher .*
unlink(p$_f$):	*management request to finalise a connection to philosopher which shares fork f with this philosopher .*
thisp:	*the unique identity of this philosopher*
other(f):	*the other fork from f that a philospher uses (e.g. other(left) = right)*

```
(R10) connection finalisation:
        unlink(p_f) =>
              fork(f):=false; reqf(f):=false; dirty(f):=false
        assert(passive)
(R11) Initialising a connection to p_f where p_f = thisp (i.e. this philosopher)
        link(p_f), p_f = thisp =>
              fork(f):=true; reqf(f):=false; dirty(f):=true
        assert(passive)
(R12) Initialising a connection to p_f where p_f > thisp (i.e. this philosopher allocates fork)
        link(p_f), p_f > thisp =>
              if fork (other(f)) then          {if the philosopher has the other fork}
                     fork(f):=true; dirty(f):=true; send(new_reqtoken_f)
              else
                     reqf(f):=true; send(new_forktoken_f)
        assert(passive)
(R13) Initialising a connection to p_f where p_f < thisp (i.e. the other philosopher allocates fork)
        link(p_f), p_f < thisp =>
              receive( new_reqtoken_f) => reqf(f):=true
         or   receive(new_forktoken_f) => fork(f):=true; dirty(f):=true
        assert(passive)
```

Rule (R10) ensures that when a connection between two philosophers is unlinked, the shared fork is removed. Rules (R11, R12, R13) are responsible for the allocation of forks when connections are linked.

Rule (R11) deals with the trivial case where there is only one philosopher which is connected to itself. In this case the philosopher is allocated two dirty forks so that it can eat. Rules (R12, R13) ensure that the global consistency requirements of a system with two or more philosophers are not violated. When two philosophers are connected together, we can satisfy fork sharing by ensuring that only one fork is allocated between them. To achieve this, we assume that each philosopoher has a unique identity and that these identities have a total ordering. The philosopher which precedes its neighbour in the total ordering decides where a fork is to be allocated. Consequently only one fork is allocated. To satisfy preservation of the acyclic precedence graph, allocation must be such that at least one philosopher ends up having two dirty forks or none. In the following, we demonstrate that rules (R12, R13) maintain an acyclic precedence graph for arbitrary changes.

Creating a Community of Philosophers

The following configuration specification describes a ring of **N** philosopher processes:

```
RING(p,N)::
      forall i:1..N create p[i];
      forall i:1..N
             link   p[i] to p[(i mod N)+1],
                    p[(i mod N)+1] to p[i];
```

The corresponding change transaction in this case is simply the specification with the added activate actions as follows:

forall i:1..N **activate** p[i].

To preserve the precedence graph invariant, the forks must be distributed asymmetrically such that at least one philosopher has no forks and correspondingly one has two forks (see initalisation conditions for the original algorithm). Rules (R12 & R13) achieve this since the identity of one philosopher must precede all others in the total ordering. This philosopher will allocate forks to both its neighbours (R12) and consequently have no forks itself.

Birth of a New Philosopher

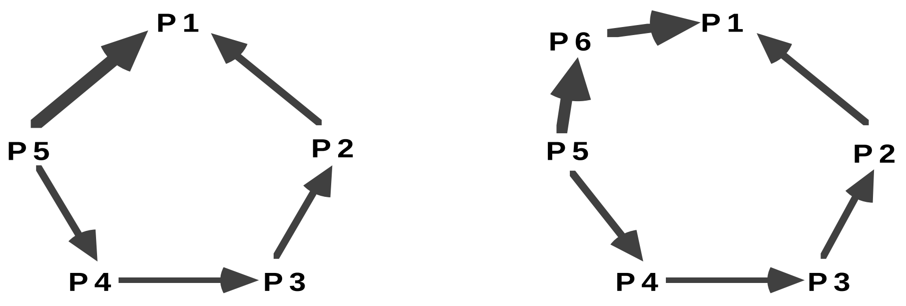
The following configuration specification adds a new philosopher **x** between existing neighbours **u** and **v**, where the other neighbours of u and v are **t** and **w** respectively **:**

```
unlink u from v; unlink v from u
create x
link x to v;  link x to u;  link u to x; link v to x
```

Applying the change algorithm of the previous section to the above change specification produces the following change transaction. From both the pre-conditions of unlink and link, the quiescent set QS is determined as the two neighbours **u**, **v** of the node to be created **x.** The change passive set is **t**, **u**, **v**, and **w**. Since neither **t** nor **w** will initiate transactions on **u** or **v** to request forks, **u** and **v** can make decisions based on the state of their forks which will not change. For example, to insert a new philosopher **P6** between **P5** and **P1** in the system depicted in Figure 5.1, **P5** and **P1** must be quiescent since they are both linked to each other and will be linked to the new posopher. The change passive set includes **P2** and **P4** as well. Note that, in the change transaction outlined below, actions on the same line may be executed in parallel.

```
passivate t,u,v,w; create x
unlink u from v; unlink v from u
link x to v; link x to u; link u to x; link v to x
activate t,u,v,w,x
```

The fork shared by u and v will be discarded when they are unlinked. On linking the pairs u and x, and x and v, the allocation of the shared fork in each case will be made by one of the pair such that one of each pair will end up with either two forks or no forks. For instance, in the example above of the addition of **P6**, **P1** and **P5** will perform the allocation as they precede **P6** in the total ordering. If the fork shared by **P2** and **P1** is currently held by **P1**, then **P1** will retain the dirty fork shared with **P6**; and if **P5** does not have the fork shared with **P4**, then it will allocate the other shared dirty fork to **P6** (Fig.5.3). This clearly preserves the acyclic graph.



(a) before addition **(b) after addition of P6**

Figure 5.3 - Addition of a philosopher

Death of a Philosopher

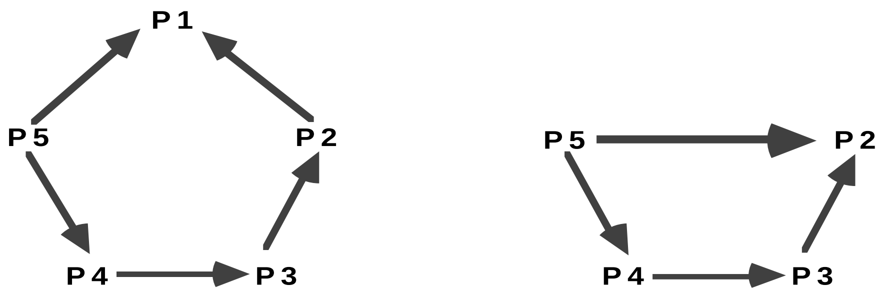
Removal of a philosopher **x** with neighbours **u** and **v** (where the opposite neighbours of u and v are **t** and **w** respectively) is specified by the following program:

```
remove x
link v to u; link u to v
```

This results in the following change transaction:

```
pasivate t,u,v,w,x
unlink u from x; unlink v from x
remove x
link v to u; link u to v
activate t,u,v,w
```

This transaction ensures that **u,v** and **x** will be in the quiescent state before **x** is unlinked and removed. Consequently, on linking, **u** and **v** can make decisions based on the state of their forks which will not change. As before, allocation will ensure that one of the pair ends up with two or no forks. For example, if in figure 5.4 we removed philosopher **P1**, **P2** would retain the dirty fork shared with **P5** as it has the fork shared with **P3**, thereby preserving acyclic precedence.



(a) Before removal. **(b) P1 removed.**

Figure 5.4 - Removal of a philosopher

Merging two communities of philosphers

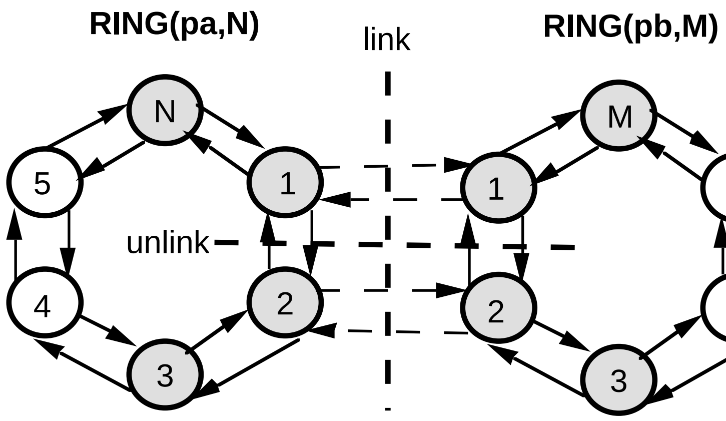


Figure 5.5 - Merging Philosopher Rings

Given two communities (rings) of philosophers, called **pa** of size N and **pb** of size M respectively, figure 5.5 shows which connections must be unlinked and which must be linked to merge two the rings of philosophers. The shaded nodes indicate the change passive set. The change is specified as follows:

```
MERGE::
        unlink pa[1] from pa[(1 mod N))+1], pa[(1 mod N))+1] from pa[1],
                pb[1] from pb[(1 mod M))+1], pb[(1 mod M))+1] from pb[1];
        link   pa[1] to pb[1], pb[1] to pa[1];
               pa[(1 mod N))+1] to pb[ (1 mod M))+1],
               pb[ (1 mod M))+1] to pa[(1 mod N))+1];
```

The corresponding change transaction is:

```
MERGE_TRANSACTION::
```

```
passivate pa[N],pa[1],pa[2],pa[3],pb[M],pb[1],pb[2],pb[3];
unlink pa[1] from pa[2], pa[2] from pa[1],
           pb[1] from pb[2], pb[2] from pb[1];
link    pa[1] to pb[1], pb[1] to pa[1,
        pa[2] to pb[2], pb[2] to pb[2]
activate pa[N],pa[1],pa[2],pa[3],pb[M],pb[1],pb[2],pb[3];
```

To justify that this change maintains an acyclic precedence graph we need only be concerned with the connection between philosphers which completes the ring. In Figure 5.5, this is performed between pa[1] and pb[1] or between pa[2] and pb[2]. Rules (R12 &R13), defined for the linking and unlinking of philosophers, ensure that the philosopher which allocates the fork on that connection retains the fork if it has the other shared fork, otherwise it allocates the fork to its neighbour. In the former case, the allocating philosopher will have two dirty forks, in the latter no forks. In fact, in the situation where neither the allocating philosopher nor its newly connected neighbour has another fork, it does not matter where the new fork is allocated since some other philosopher must have two forks. This can be easily argued as follows:

> There are n philosophers and n forks; the two philosophers being connected have 1 fork, consequently the remaining n-2 philosophers have n-1 forks. Therefore, one of these n-2 philosophers must have 2 forks. The original algorithm ensures that a philosopher cannot hold a clean and a dirty fork simultaneously; consequently, the precedence graph must be acyclic.

Note that inserting a new philosopher into an existing ring of philosophers is equivalent to merging a ring of one philosopher, RING(newphil,1), with an existing ring. Splitting a ring into two smaller rings requires a change specification opposite to that of MERGE. As before, the connection which completes each ring preserves the global invariant.

This section has shown how the management protocol is applied to a specific application. Only those philosophers in the change passive set are affected by the change allowing the rest of the system to proceed with its normal execution. Changes can be carried out in parallel as the stable states ensured by node quiescence permits consistent decisions to be made during linking and unlinking. The different cases descibed in this example have been prototyped and validated in the Conic environment for distributed programming [14,15,19]. Further work is required to integrate the change management states into the current environment.

6. DEPENDENT TRANSACTIONS

In the discussions above, we have considered only two party independent transactions. We now relax the restriction of independence and discuss systems using **dependent** transactions, which involve one or more consequent transactions. In general, systems include both independent and dependent two party transactions.

• A **dependent** transaction is a two-party transaction whose completion may depend on the completion of other **consequent** transactions.

This is described more precisely as follows: t_i is a dependent transaction if there exists a chain of transactions t_i, t_j, ...t_s in which each, with the exception of t_s, may depend for completion on the completion of its (consequent) successor transaction. We do not forbid cycles in this chain, but require that:

(i) progress is made to ensure that the transaction is still bounded,

(ii) that deadlock is avoided (for example, that it is not a cycle of nested transactions).

We require that the initiator of a dependent transaction is informed of the completion of consequent transactions. This enables a node to determine when transactions, which it initiated, have completed and hence when it has achieved a passive state.

Extension of the Independent Transaction Approach

Consider a number of client nodes Ci which access a printer server S via their agent Ai and a server manager node M (Figure 6.1). In this case, a transaction may consist of a sequence of message interactions involving Ci, Ai, M and S. For instance, C1 may initiate transaction s1 to request a print service; completion of s1 is dependent on the consequent transactions r1 and p, which A1 and finally M will initiate to S, to actually print the lines. Dependent transactions and their potential consequent(s) are denoted as **dependent/consequent(s),** as illustrated in figure 6.1 where si is dependent on ri, which is dependent on p.

The change transaction discussed in relation to independent transactions, such as removal of S, would require that M is quiescent and all agents Ai, M and S are in the passive state. This implies that S could be removed when M and A complete their current two party transactions ri and p, and S completes the associated processing. However, since Ci may still have further lines to send (si is not complete and may require further ri and p transactions), this is clearly not sufficient to maintain system consistency.

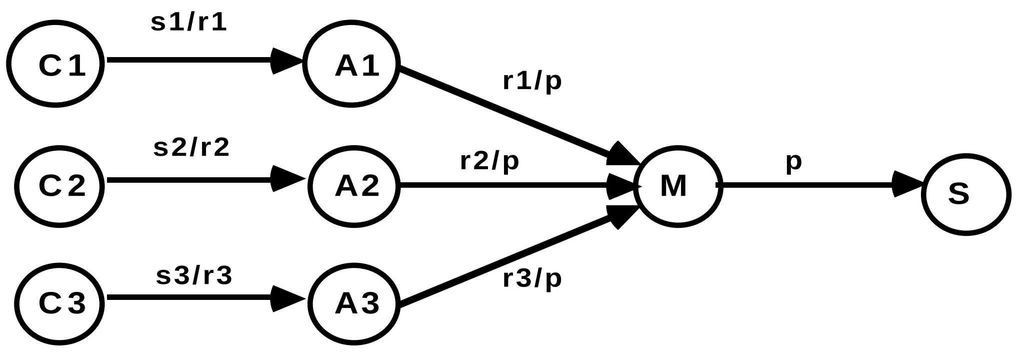

$$\begin{tikzcd}
C1 \arrow[r, "s1/r1"] & A1 \arrow[dr, "r1/p"] & & \\
C2 \arrow[r, "s2/r2"] & A2 \arrow[r, "r2/p"] & M \arrow[r, "p"] & S \\
C3 \arrow[r, "s3/r3"] & A3 \arrow[ur, "r3/p"'] & &
\end{tikzcd}$$

Figure 6.1 - Client-Agent-Manager Interaction using Dependent Transactions

Hence, if the change passive set of nodes consists only of the set of nodes which can initiate transactions on a target quiescent node plus the node itself, there is no guarantee that the node is in the quiescent state. Nodes may still initiate consequent transactions on the target node to satisfy completion of arriving transactions. There are two main approaches to providing quiescence in such systems with dependent transactions:

1. Require that consequent transactions are recognised in the application.

M must recognise that the transactions with S are part of and a consequence of a wider transaction, and that the passive state can only be reached when that client transaction completes (i.e. M must wait for completion of the client use of the printer before it can become passive). This approach has the disadvantage of having to embed, in the application, knowledge of all the transaction dependencies. In addition, this information would be hidden from the configuration view. Transactions would implicitly ripple back and require some time to complete, although we would still require that they do so in bounded time. Another possibility is to abort transactions with consequents when in the passive state. For example, M could abort the client transaction with C. This carries the overhead of complicating C such that it must regain consistency after transaction abortion (cf. recovery). Aborting reduces dependent transactions to independent transactions at the expense of complicating the application code.

2. Expand the passive set to include all nodes initiating dependent transactions which result in transactions on the link or node targeted for change .

This requires that transaction dependencies are reflected up to and taken account of at the configuration level. Since it is at the configuration level that we wish to manage changes, this is the preferred approach.

Generalised Passive State for Systems using Dependent Transactions

The proposed approach for systems using dependent transactions is to expand the passive set PS(Q) to include those nodes which initiate transactions which have consequent transactions on the node Q required to be quiescent. However, the passive state may not be reachable for nodes utilising dependent transactions. Consider the example in figure 6.2. In this system suppose **N3** is in the passive state and **N1** has initiated transaction **a.** In this situation transaction **a** cannot complete because transaction **b** cannot complete because N3 cannot initiate **c**. Consequently, neither **N1** or **N2** can move into the passive state in bounded time if requested. Hence, ***PROPOSITION 1 does not hold for dependent systems.***

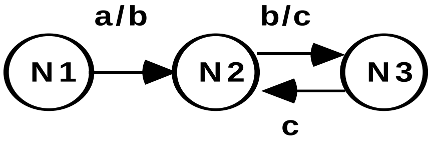


Figure 6.2 - A system with cyclic dependencies.

We could consider providing an **ordering** on the passive set such that nodes are made passive in the order of the dependence graph. In the example in figure 6.2 we would passivate in the order N1 then N2 then N3. However, this order cannot be determined in general. In the example in figure 6.3, transaction a/b requires N1 before N2, whereas c/d requires N2 followed by N1.

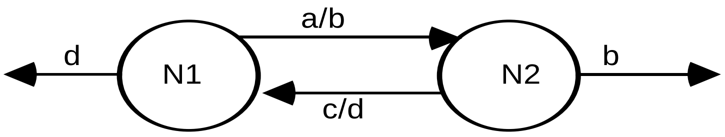


Figure 6.3 - A system with mutual dependencies.

A more appropriate approach is to generalise the definition of the passive state to include the means for dependent transactions to complete:

• A node in the **general passive** state must accept and service transactions ***and initiate consequent transactions***, but
(i) it is not currently engaged in a (non-consequent) transaction that it initiated, and
(ii) it will not initiate any new (non-consequent) transactions.

Thus a node in the general passive state must respond to transactions while it is in the passive state, **and**, it must initiate any consequent transactions required for the completion of the transactions to which it responds. For independent transactions, the definition of the general passive state reduces to the same as passive.

For example, in figure 6.4, if node **N** is in the general passive state it must not initiate **x** or **y** as a result of responding to **b** or **c**. However, it may initiate **x** to permit completion of **a**. (For conciseness, we henceforth use *passive* to mean *general passive* where such use is unambiguous).

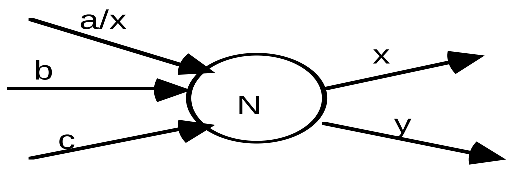


Figure 6.4 - Independent and Dependent transactions on node N.

Resultant Properties for Systems using Dependent Transactions

PROPOSITION 1' (Reachability of the Passive state)
Given the generalised definition of the passive state, Proposition 1 holds for systems using both independent and dependent transactions.

JUSTIFICATION 1':
As for proposition 1, with the added justification that dependent transactions will also complete in bounded time even if the recipient nodes are in the (general) passive state since they respond to and can initiate consequent transactions.

The definition of **quiescence** for systems using dependent transactions remains the same as that for independent transactions (see section 3). However, as discussed above, the passive set must be expanded in order to account for dependent transactions which lead to consequent transactions on the node.

- The **enlarged passive set *EPS*** for a node **Q, EPS(Q),** is defined as follows:
  1) all nodes in **PS(Q)** are in ***EPS*.**
  2) all nodes which can initiate dependent transactions which result in consequent transactions on **Q** are in ***EPS***.

PROPOSITION 2': (Passive requirements for the Quiescent state)
Given the generalised definition of the passive state and the enlarged passive set, Proposition 2 holds for both independent and dependent systems (i.e. if node N and all nodes in the enlarged passive set with respect to N are passive, then N is quiescent).

JUSTIFICATION 2':
As for proposition 2, with the added justification that all nodes which can initiate transactions (independent, dependent or consequent) on N are passive, then all transactions involving N will be complete.

PROPOSITION 3': (Reachability of the Quiescent state)
Given the generalised definition of the passive state and the enlarged passive set, Proposition 3 holds for both independent and dependent systems.

JUSTIFICATION 3':
Follows directly from propositions 1' and 2'.

Change Rules

The change rules remain as before, except that the region of quiescence where the change is to occur results in an enlarged set of passive nodes specified by the EPS.

COMPOSITION RULES

In the foregoing, we have been concerned with flat or one-level graphs of connected nodes. However, in general, we are concerned with an hierarchic graph structure such that nodes at one-level may themselves be implemented as graphs of connected nodes at the next level of detail. For example, in the Conic system which represents systems as configurations of logical nodes, these logical nodes are themselves implemented as a graph of subnodes or tasks. The Conic logical node is the unit of change and allocation, and the task is the unit of concurrency [19]. To ensure that the change management system need be concerned with only one level of the configuration graph at a time we must be able to derive the transaction dependency relations of a node from those of its subnodes. In the following, a node which is composed of subnodes is referred to as a composite node. A substitution rule can be used to determine the dependencies of composite nodes from the dependencies of their constituent nodes:

Node Composition by **substitution**:
In composing two nodes, substitute the consequents for each occurrence of the dependent transaction which is hidden by the composition (see figure 6.5)

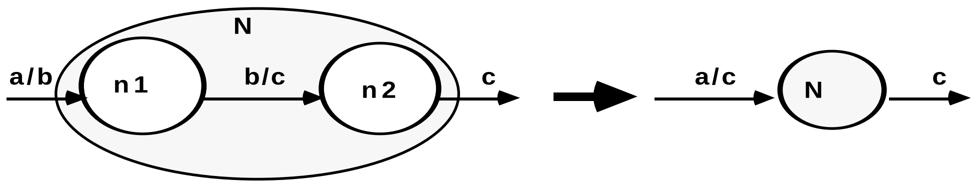


Figure 6.5 - Derived dependencies for simple node composition .

For more complex structures, the rule can be used for each connection and by repeated application for each node composition. For example, consider the fork structure in figure 6.6, where a is potentially dependent on b and/or c.

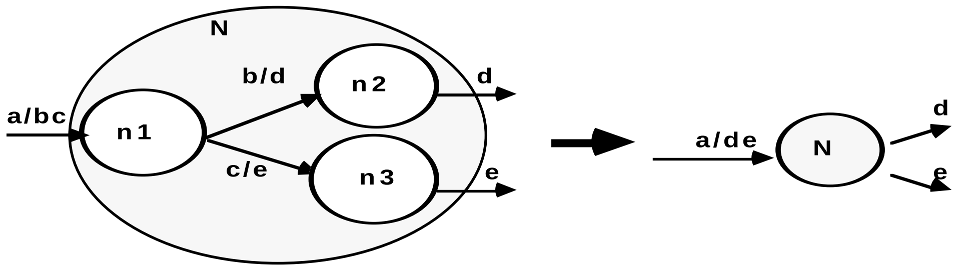


Figure 6.6 - Derived dependencies for repeated node composition .

Internal transactions are not visible in composite nodes. For example, each philosopher node of the Evolving Philosophers problem can be (and was) implemented as a composite node as shown in figure 6.7. This solution structure follows that of [1] (which addresses only the original Dining Philosophers problem).

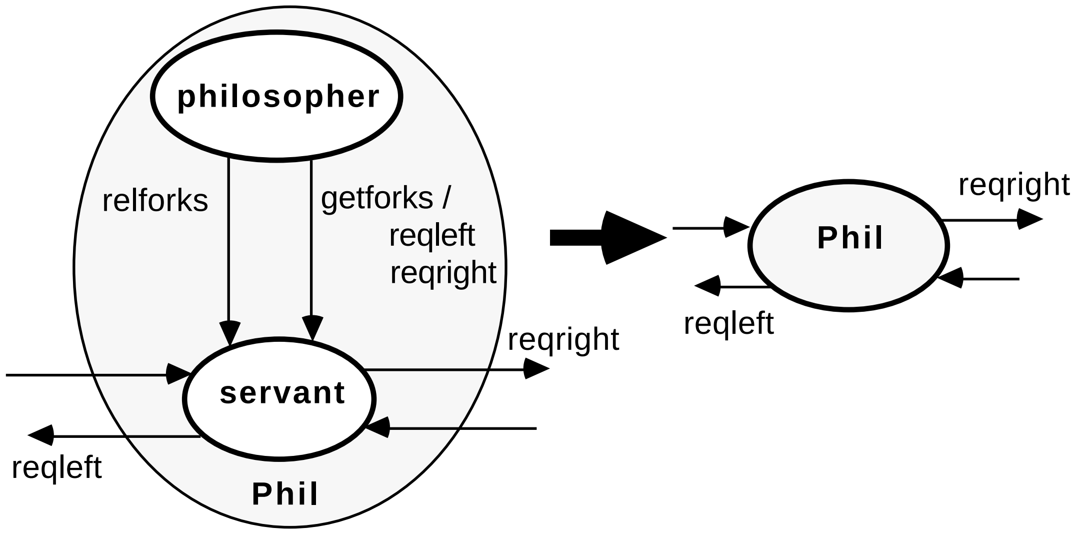


Figure 6.7 - Composite Philosopher Node

The philosopher subnode implements a simple state machine to control the transitions between thinking, hungry and eating, while the servant subnode encapsulates the protocols necessary to acquire forks. The internal transactions *relforks* and *getforks* are not visible in the composite node *Phil.* It should also be noted that the dependency of the philosopher transaction *getforks* on the consequent transactions *reqright* and *reqleft* (the transactions to request a left and a right fork) is not visible in the composite node. However, this dependency means that for the node *Phil* to be passive both its subnodes (*philosopher* and *servant*) must be passive.

In our prototype implementation of change management in Conic, we have adopted the following simplified but pragmatic strategy. The change management system views the system as a one-level graph of logical nodes. As mentioned above, logical nodes are both the unit of change and the unit of allocation in Conic. Logical nodes are constrained by design to communicate by independent transactions so that the management system need not be aware of dependency information. The structure of a logical node is fixed at node instantiation time. Transactions between subnodes can be independent or dependent as shown in figure 6.7. To simplify local node management we have implemented the rule that a logical node is passive when *all* its constituent nodes are passive. A local entity collates management state for each logical node.

Composition thus provides a coarser grain for system configuration and dynamic change management. For a finer grain, decomposition can be used (where appropriate) to expose the internal structure of connected nodes to make them accessible to change.

7. DISCUSSION AND CONCLUSIONS

This paper has presented a comprehensive model of change management which clearly separates the management responsibilities and view from that of the application (see fig. 1.1). In particular, the objectives given in section 2 have been approached as follows:

Changes should be specified in terms of the system structure.

Changes are specified in terms of the primitives **create**, **remove**, **link** and **unlink** which refer only to system strucure. In fact, changes specified in these primitives could be derived from the difference between specifications of the current and desired system structures.

Change specifications should be declarative.

It is the derived change transactions which include the change control actions (**activate** and **passivate**) and specify the parallelism or sequencing of the actual change execution.

Change specifications should be independent of the algorithms, protocols and states of the application.

The node configuration states (**active**, **passive**) abstract away the specific application states, and provide a convenient means for viewing and controlling the application.

Changes should leave the system in a consistent state.

The passive and quiescent states together with the possible inclusion of connection initialisation and finalisation code provide the application with the means to preserve application consistency in a convenient and pragmatic manner.

Changes should minimise the disruption to the application system.

The change passive set identifies the set of nodes affected by a change. In fact, the set is not currently minimal. For instance, in the client server example in section 4, the clients need only be passive with respect to the particular server being removed and need not actually be prevented from initiating transactions on other (unillustrated) nodes.

Let us consider the connection level further. The current model provides for change in the form of creation and deletion of nodes, and connection changes. Change which only involves connections could place emphasis on each connection rather than the entire node which initiates that connection. Thus the state of each connection (disconnected or connected-passive or connected-active) could be modelled, together with the consistency preserving actions associated with each connection. This leads to a finer grain model in which a node can be active with respect to one connection yet passive with respect to another. This approach appears to be promising in its ability to describe connection changes at a finer level of granularity however, it does require more passive substates. Furthermore, the design of the connection level actions seems to be more difficult since the node and its environment may be partially active thereby making consistency more difficult to attain. Hence, although our current approach of requiring complete node quiescence may not be minimal in terms of the disruption to a system, it does seem to be sufficient and far simpler to reason about and use.

Dependent and Independent Transactions

The approach adopted for dependent transactions generalises the passive state of a node to permit initiation of consequent transactions and enlarges the passive set to include nodes which can initiate dependent transactions with consequents on the nodes previously in the set. This expansion of the passive set corresponds to our intuition that changes to systems which are more interdependent require more global quiescence and cause more disturbance (i.e. close-coupling makes change more difficult). We believe that the model confirms and, to some extent, quantifies that interdependence. One approach to alleviating this interdependence, is to compose dependent nodes together so that composite nodes communicate using only independent transactions. Changes must then be performed at a 'coarse grain' level on composite nodes rather than on constituent dependent nodes. Alternatively, dependent systems can be reduced to independent systems for the purpose of management, if transactions can be aborted by passive nodes. The cost of this is the extra complexity incurred by the application to preserve consistency in the presence of aborted transactions (c.f. atomic transactions). However, in real systems, this cost may be inevitable to deal with failure.

Detection of the Passive state

The transactions used in the model require that the initiator is aware of the completion of the transaction, whether it is dependent or independent (see definition in section 2). This is required in order that a node can determine when outstanding transactions that it initiated are complete and hence when it is passive. If this were not the case, it is possible that a node could assume completion of a transaction which was actually delayed in communication and still outstanding. This requirement could be relaxed to permit, for instance, asynchronous messages if the node or management system used some other method for detecting termination of transactions. This would require use of a distributed termination algorithm such as the diffusing detection algorithm of Francez [8]. Assuming that messages do not overtake one another, the management system could initiate detection by sending queries along the dependency chains of the nodes in the passive set and obtaining confirmation if all nodes agree that they are indeed passive.

Related Work

The changes described in this model are directed at the operational system itself, in terms of changes to the software components and their interconnections. It can be contrasted with the model for change incorporated in the Inscape Environment [21] which concentrates on change validation in relation to a static definition of the system. Inscape utilises a semantic interconnection model which could form a useful adjunct to our model by permitting static change validation before application to the system itself. A promising and related approach which could be used to model and analyse dynamic configuration changes has been suggested using graph grammars (Garp [11] and Δ-Grammars [12]) . This provides a formal graphical description of system structure which is equivalent to our configuration specification. Changes are specified in terms of Δ transitions which act on the system structure to produce new structures. However, unlike our approach, they have chosen to model aspects such as message passing at the structural level, thereby making the specification of changes rather more complex than ours at the configuration level. Also, their model appears to be purely for specification purposes, and gives no indication as to how it might be realised. For instance, it is not clear how detailed consistency constraints, such as those preserved by the actions in the evolving philosophers, could be modelled in Δ-grammars.

Pragmatic approaches to dynamic change management have tended to concentrate on code replacement. The most simple strategies are little more than traditional object code patching which relies on recovery to ensure system consistency. More recently Frieder and Segal [9] have suggested a scheme for procedure replacement which does not require recovery. While we ensure that component quiescence will occur, they rely on detecting procedure quiescence before performing a change. Continuously active procedures can thus not be replaced in their scheme. Further while we are concerned with arbitrary restructuring of a system their scheme is firmly focussed on replacement.

The transformational approach [2] advocates that changes should be dealt with at the formal specification level. The new system is then 'regenerated' from that changed specification using transformational techniques. However, in order to avoid regenerating the entire system, the changed parts need to be identified and generated. Also, dynamic introduction of the changes to an operational system would still need to be supported in some way. Given that even the transformational aproach needs to describe non-trivial systems as some composition of components, our model provides a means for obtaining the required structural changes from the new structural specification (Figure 1.2) and of deriving the change transactions for integration of the changes dynamically. Hence, although at first sight the two approaches appear to be incompatible, our model provides a systematic and pragmatic basis which could be used in conjunction with the transformational approach.

Further Work

The paper has concentrated on evolutionary change where change is instigated by an agent external to the system. However, the change protocol can equally be invoked internally by the application. The application can minimise the disruption caused by a change by instigating the change when quiescence is detected rather than externally imposed.

Change could also occur as a result of failure. Although not explicitly handled by the model, we believe that failures can be handled if the nodes incorporate the necessary recovery actions. These would be used to restore the remaining system to consistency in conjunction with the reconfiguration actions which could be triggered by detection of failure. This area requires further investigation.

The management of evolutionary change is a difficult but important issue. It is therefore essential that the techniques adopted are both practical and soundly based. We believe that our approach, with its clear separation of structural management and application concerns, is very promising in these regards. Some small case studies have been prototyped and tested in the Conic environment for distributed programming which provides both textual and graphical facilities for performing dynamic configuration changes [16]. It now remains to be further refined, formalised and tested on larger case studies.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues Morris Sloman, Naranker Dulay, Kevin Twidle and Keng Ng during the formulation of these ideas. In particular we acknowledge the contribution of Andrew Young in prototyping and refining the change management model. We gratefully acknowledge the SERC ACME Directorate under grant GE/E/62394, and the CEC in the REX Project (2080) for their financial support .

REFERENCES

[1] ANDREWS, G.R ET AL. "An Overview of the SR Language and its implementation", *ACM TOPLAS*, Vol 10, No. 1, January 88, pp. 51-86.

[2] BALZER, R. "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering*, SE-11 (11), Nov. 1985.

[3] BARBACCI, M.R., WEINSTOCK, C.B., AND WING, J.M. "Programming at the Processor-Memory-Switch Level", *Proc. IEEE 10th Int. Conf. on Software Eng.*, Singapore, April 1988, pp 19-29.

[4] BLOOM, T. "Dynamic Module Replacement in a Distributed System", Technical Report MIT/LCS/TR-303, MIT Laboratory for Computer Science, March 1983.

[5] CHANDY, K.M., AND MISRA, J. "The Drinking Philosophers Problem", *ACM TOPLAS*, Vol.6, No.4, October 1984, pp. 632-646.

[6] DE REMER, F., AND KRON, H. "Programming-in-the-large versus Programming-in-the-small", *Proc. Conf. Reliable Software*, 1975, pp. 114-121.

[7] DIJKSTRA, E.W., "Hierarchical Ordering of Sequential Processes", in Operating Systems Techniques, C.A.R.Hoare and R.H.Perrot (Eds.), Academic Press, 1972.

[8] FRANCEZ, N. "Distributed Termination", *ACM TOPLAS*, Vol.2, No.1, January 1980, pp. 42-55.

[9] FRIEDER, O. AND SEGAL, M. E., "Dynamic Program Updating in a Distributed Computer System", *Proc. IEEE Conf. on Doftware Maintenance,* Phoenix, Arizona Octber 1988, pp 198-203.

[10] FRIEDBERG, S.A. "Transparent Reconfiguration requires a Third-Party Connect", TR220, Computer Science Department, University of Rochester, New York, Nov. 1987.

[11] KAPLAN S.M., KAISER G.E., "Garp: Graph Abstractions for Concurrent Programming", *ESOP '88,* Nancy, France, March 1988, Springer-Verlag, pp. 191-205.

[12] KAPLAN S.M, GOERING S.K., CAMPBELL R.H., "Specifying Concurrent Systems with Δ-Grammars", Proc. of 5th International Workshop on Software Specification and Design, Pittsburgh, May 1989, pp. 20-27.

[13] KRAMER, J., AND CUNNINGHAM, R.J. "Towards a Notation for the Functional Design of Distributed Processing Systems", *IEEE Proc. 1978 Int. Conf. on Parallel Processing*, Aug. 1978, pp. 69-76.

[14] KRAMER, J., AND MAGEE, J. "Dynamic Configuration for Distributed Systems", *IEEE Transactions on Software Engineering*, SE-11 (4), April 1985, pp. 424-436.

[15] KRAMER, J., MAGEE, J., AND SLOMAN, M. "The CONIC Toolkit for Building Distributed Systems", *IEE Proceedings*, Vol. 134, Pt. D, No.2, March 1987, pp 73-82.

[16] KRAMER, J., MAGEE, J., AND NG, K. "Graphical Configuration Programming", IEEE Computer, Vol.22, No. 10, October 1989, pp 53-65.

[17] LEBLANC, T.J., AND FRIEDBERG, S.A. "HPC: A Model of Structure and Change in Distributed Systems", *IEEE Trans. on Computers*, Vol. C-34, 12, December 1985, pp 1114-1129.

[18] LEHMAN, M.M. "Program Evolution", *Proceedings of Symposium on Empirical foundations of Information and Software Science*, Atlanta, Georgia, Nov. 1982.

[19] MAGEE, J., KRAMER, J., AND SLOMAN, M. "Constructing Distributed Systems in Conic", *IEEE Trans. on Software Engineering*, SE-15 (6), June 1989, pp. 663-675.

[20] MISRA, J. "Reasoning about Networks of Communicating Processes", INRIA Advanced

NATO Studies Institute on Logics and Models for Verification and Specification of Concurrent Systems, Nice, France, 1984.

[21] PERRY, D. E. "Software Interconnection Models", Proc. of 9th Int. Conf. on Software Engineering, April 1987, pp 142-149.