Reliable Network Communications

Gail E. Kaiser
Yael J. Cycowicz
Wenwey Hseush
Josephine Micallef
Columbia University
Department of Computer Science
New York, NY 10027

December 1987

CUCS-278-87

Abstract

This technical report consists of three papers from the INTERCOMS project. *A Network Architecture for Reliable Distributed Computing* introduces the view section model, a network layer for exception handling in response to disruptions in communication channels due to failures of network links or nodes. *Remote Exception Handling* discusses for a network layer for exception handling among cooperating application processes. *Demand-Driven Parameter Passing in Remote Procedure Call* describes how remote exception handling solves the problem of passing referential data types (pointers) as parameters to remote procedures.

This research is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. Ms. Cycowicz is an AT&T Fellow. Mr. Hseush is supported in part by the New York State Center of Advanced Technology — Computer & Information Systems. Ms. Micallef is a past IBM Fellow.

Reliable Network Communications

Gail E. Kaiser
Yael J. Cycowicz
Wenwey Hseush
Josephine Micallef
Columbia University
Department of Computer Science
New York, NY 10027

December 1987

Abstract

This technical report consists of three papers from the INTERCOMS project. *A Network Architecture for Reliable Distributed Computing* introduces the view section model, a network layer for exception handling in response to disruptions in communication channels due to failures of network links or nodes. *Remote Exception Handling* discusses for a network layer for exception handling among cooperating application processes. *Demand-Driven Parameter Passing in Remote Procedure Call* describes how remote exception handling solves the problem of passing referential data types (pointers) as parameters to remote procedures.

This research is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. Ms. Cycowicz is an AT&T Fellow. Mr. Hseush is supported in part by the New York State Center of Advanced Technology — Computer & Information Systems. Ms. Micallef is a past IBM Fellow.

A NETWORK ARCHITECTURE FOR RELIABLE DISTRIBUTED COMPUTING

Wenwey Hseush Gail E. Kaiser

Columbia University
Department of Computer Science
New York, NY 10027

Abstract

The complexity of message passing in loosely-coupled distributed systems is dramatically increasing, partially due to the movement towards large scale distributed systems and intelligent distributed applications. Traditional approaches such as the client-server model are no longer appropriate. We propose a *reliable distributed environment* (RDE) based on an efficient and reliable extension to datagram communications that provides reliable communication and configuration services. We introduce the *coupled relation* to measure the degree of reliability of distributed environments. We also present *view sections*, which protect against changes in node status (available or unavailable) in the same sense that *critical sections* protect against changes to shared memory, as support for distributed communications tasks. We give simulation results for coupled relations based on different algorithms, node failure rates, recovery times and message arrival rates, and to illustrate the behavior of distributed systems constructed using our view section model on top of RDE.

1. Introduction

A *Reliable Distributed Environment* (RDE) is a collection of loosely-coupled distributed nodes where the environment ensures *reliable communication* and *close view*. Reliable communication guarantees messages are received by the destination nodes if the destination nodes are functionally working at the moment of message arrival, thus protecting against link failures. Close view provides a snapshot of the environment to protect against node failures (process deaths, machine failures[1], or any temporary functional failures on nodes). Close view implies precise prediction of node status. We use the term "node" to refer to a "process" in the transport layer in order to distinguish from a "host" in the network layer and the term "virtual circuit" to refer to an end-to-end communication channel in the transport layer.

It would be unnecessary for us to propose RDE if the complexity of message passing had remained as simple as in most traditional distributed applications, to which the client-server model[2, 1] has been applied successfully. The client-server model implies end-to-end communications between two different nodes which need not know the status of any nodes except each other. The notion of close view is becoming important since the complexity of message passing is dramatically increasing in cases like large scale and/or intelligent distributed systems, which both require more complicated communications patterns. The client-server relationship no longer holds and failure to predict network-wide node status results in severe degradation of performance.

We propose a programming framework, the *view section model*, in which to construct reliable distributed computing tasks on top of RDE. View sections protect against the change of the global view as *critical sections*[3] protect against the change of shared memory. A *view section* defines a period of time and a sequence of instructions during which the global view should remain the same to maintain the correctness of the computation performed by the instructions. The fact is, however, that the global view changes from time to time as nodes fail and are restored, even during view sections. We handle this by invoking an application-specific *compensation function* via an immediate notification generated by RDE when it senses a change of the global view. The compensation function decides what to do to preserve the view section. Further work is required to construct a full transaction mechanism[4, 5] based on the view section; note therefore that we are not concerned here with the issues of reliable distributed databases.

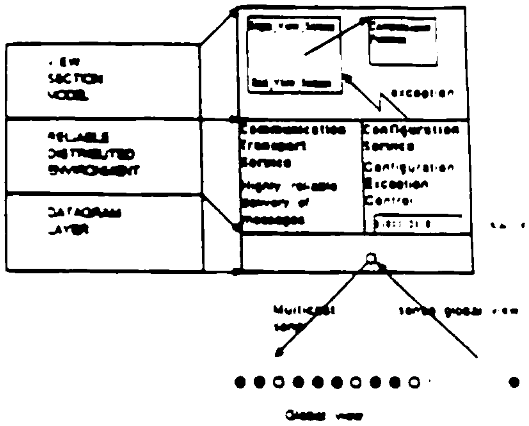


Figure 1. A Network Architecture For Reliable Distributed Computing

1.1 Simple Communication Patterns

In most loosely-coupled distributed systems[6], typical communication patterns are limited to the following cases so that the existing transport layer protocols (*e.g.*, TCP[7]) can perform efficiently and reliably. We use the term "communication task" to refer to a collection of message passing interactions between two or more nodes.

1. Client-server model: A temporal and reliable virtual circuit or an end-to-end datagram link is built between two nodes, say nodes C and S, at the beginning of a communication task and is disconnected when the task is finished in order to permit following communication tasks, associated with either node C or S, to be processed.

2. Complete connections in small domains: Node domains, which specify nodes in distributed systems, are so small and fixed that a complete set of virtual circuits can be built for message communications during system initialization, and the virtual circuits can be closed upon the termination of the distributed system.

1.2 Complicated Communication Patterns

In large scale and intelligent distributed systems, communication patterns are much more complicated than these conventional models can handle. Consider the following cases:

1. Single-interaction communication tasks: Single communications randomly and frequently take place among the possible pairs of nodes. Typically, instead of using datagram, virtual circuits are applied to end-to-end communications because of the necessity of reliable message delivery. The datagram protocol neither guarantees that messages are safely received by the remote node nor notifies the sender what happened on the remote node. Unfortunately, it is inappropriate to use virtual circuits for end-to-end communications in this case, because the cost of connecting reliable virtual circuits is comparatively heavy against that of passing messages. The performance degrades dramatically.

2. Complete connections in a large domain: Large scale distributed systems, with a very large number of nodes, usually run in a dynamic configuration[8]. Nodes come and go without affecting the rest of the system. Again, it is inappropriate to use virtual circuits in this case, because the complexity of number of I/O ports, O(N), complicates each node. The complexity of O(N) looks reasonably good, but in fact it is bad since I/O resources are limited in most operating systems. For example, in Berkeley Unix™ the number of file descriptors which can be associated with I/O ports (*i.e.*, stream sockets[9, 10]) is quite small. For each file opened as an I/O channel, the kernel reserves a memory buffer for storing incoming data packets and pays attention (CPU time) to detecting the failures of connections and nodes. These costs are expensive.

3. Multicast transport services: Some distributed systems initiate and complete communication tasks through multicasting. Grouping, reliable message delivery and exception control are important to build a reliable distributed system. Unfortunately, in most existing transport layer network environments, multicast transport services are not supported. Even when supported in a network environment, unreliability is usually a problem.

1.3 Using Datagram For Complicated Communication Patterns

Datagram eliminates the problems of high I/O ports complexity and expensive virtual circuit connections, so it seems to be a better approach for large scale loosely-coupled distributed systems. The disadvantages and advantages of using datagram have to be pointed out to explain our design of RDE using datagram. For most existing datagram transport services (*e.g.*, User Datagram Protocol), unreliability is the major problem. Datagram packets may be delivered multiple times or out of sequence, or not delivered at all. A sender neither knows the status of the destination nodes, nor can it be assured that the message packets have been safely received. A positive acknowledgment scheme is often used to ensure safe delivery of messages. The most pleasant aspect of datagram is that only one I/O port is needed for each node to send and receive messages, which is particularly important when I/O resources are limited. Therefore, it makes more sense to implement a protocol on top of datagram to ensure reliable delivery of messages than to implement a protocol on top of virtual circuit to reduce the number of ports used.

The simplicity of one I/O port is a very convincing reason for large scale distributed systems to use datagram. This results in the trend for intelligent distributed applications to handle increasingly complex patterns of message communications using one I/O port for each node, which listens to or talks to all other nodes without building end-to-end connections and expecting a small degree of unreliability.

2. A Reliable Distributed Environment Using Datagram

The goal of a reliable distributed environment is to extend the reliability between a pair of nodes, which has been promised with virtual circuit, to the reliability among a group of nodes. As mentioned above, a reliable distributed environment is defined as a collection of loosely-coupled distributed nodes that ensures reliable communication and close view. Each distributed node has a static view, called local view, to reflect the status of the environment (*i.e.* which node is up and which node is down). Reliable communication and close view are not mutually independent. Reliable communications together with the effect of close view ensures highly reliable delivery of messages. Also, close view with the effect of reliable communications ensures precise prediction of node status and immediate notification of exceptions.

Reliable delivery means two things: i) reliable transmission of messages and ii) messages safely received by the destination nodes. Remember that 100% reliable delivery, even through a 100% reliable communication channel, is impossible, because the local view can reflect at best the environment status in the near past due to the nature of message passing, and there is no way to guarantee that messages will be safely received by the destination nodes at the moment of transmission. Our simulations, which we describe later, demonstrate that the closer the relation among nodes, the more reliable is message delivery. That is, the more precisely a node can predict its *local view* of the status of all the other nodes, the less exceptions due to unexpected events regarding message deliveries.

Consider, for example, that the positive acknowledgments required to ensure safe arrivals when multicasting a message are expected at a very high probability because of precise prediction of the global view. The basic reason why multicast requires waiting for all acknowledgments, or until timeout, is to know what is going on with the destination nodes, even though the result comes out almost the same as that predicted at the beginning of multicasting. It is unreasonable to thus sacrifice performance if the extremely small probability of exceptions can be compensated for in some way. *Highly reliable delivery* in RDE leads to high performance for the targeted distributed computing tasks. One more important service in RDE is *configuration exception control*, which is proposed to complement highly reliable delivery; this is discussed in the next subsection. Basically, the philosophy of the RDE model is highly reliable delivery plus configuration exception control. The predicting algorithms are also discussed shortly.

Let's define some terminology before we go on. *Configuration-bits* is defined as a bit string which indicates the status of nodes, active or inactive, in a designated order according to the nodes domain of the distributed environment. Each node has a *local configuration-bits* as its local view to keep track of its knowledge of the configuration of the distributed environment; each node is designated as active or inactive. *Local configuration-bits* is a special representation of local view. *Global configuration-bits*, representing the global view, is an imagined *configuration-bits* constructed from the status of all nodes in the same order as the *local configuration-bits*. *Configuration-bits* is the simplest version of local view.

2.1 RDE Services

Two important services are supported in RDE: i) *communication transport service* and ii) *configuration service*. Communication transport service supports reliable communication, and configuration service is for close view.

Communication transport service uses the standard mechanisms of sequence number and timeout/retry to eliminate the possibilities of duplication, out of sequence and missing data packets in datagram communications. It supports three types of operations:

- Multicast: send messages to a group of nodes. Nodes in the domain can be arbitrarily grouped by setting different channels. Grouping will be discussed with configuration service below.
- End-to-end send: send message to one node. The two relevant function calls for this type are *sendto* and *reply*.
- Multiread: read messages from multiple inputs. Two or more I/O ports can be created for different classes of communications. Each I/O port corresponds to a domain. Two or more domains can be specified in a distributed environment. Each domain defines a class of communication.

The major difference between RDE and traditional transport services is that the messages are assumed to be safely received by the destination nodes at the moment of sending messages, and control is immediately returned to the caller. This is because messages are safely delivered with very high probability, which we explain in the section on simulation. The difficulties of highly reliable delivery of messages are solved by configuration exception control.

Configuration service supports the following:

- Configuration exception control: The idea is whenever RDE senses changing of configuration or unexpected conditions of message delivery, it notifies the higher level layer with a configuration exception. The service protocol guarantees that the related nodes will be notified in time t after a message delivery is initiated if the related exception occurs. The notification procedure first enters exception events into a global event queue, then generates a signal that invokes a handler routine.

- Grouping: Nodes can be grouped by setting channels. One node might become a member of multiple groups by setting two or more channels. A node can release group membership by unsetting the channels.
- Dynamic configuration: Nodes can come or go without affecting the whole system. This is very important in large scale distributed systems which disallow turning off all the nodes in order to reconfigure the environment.

2.2 Coupled Relation and Idealized RDE

In order to measure the degree of reliability of a distributed environment, the *coupled relation* is introduced as a function of how closely nodes must interact. The "more closely" the distributed nodes interact, the more reliably the distributed computing tasks are achieved. In other words, the coupled relation is used to quantify the reliability of a message-passing based distributed environment, where the reliability lies between closely-coupled and loosely-coupled with respect to message passing rather than concurrent processing. That is we try to quantify the unreliability due to the deficiency of message passing. A closely-coupled distributed environment, where nodes communicate with each other through shared memory, can be considered 100% reliable with respect to message passing.

Two versions of coupled relation have been defined: i) as the coefficient of statistical correlation between *local configuration-bits* and *global configuration-bits*, and ii) as the probability that *local configuration-bits* match *global configuration-bits*. There is a monotonically increasing relation between the coefficient of correlation and the probability. We simply use the second version as our experimental coupled relation on the simulation. This means that the coupled relation specifies how precisely a node's local view predicts the status of other nodes. If the coupled relation is equal to 1, we call it a closely coupled relation. In our RDE model, where the difficulties of highly reliable delivery of messages are solved by configuration exception control, a higher coupled relation that more precisely predicts node status will reduce the cost of exception handling.

Many parameters like message arrival rate, node failure rate, node recovery time, predicting algorithms and data missing rate affect coupled relation; these are discussed in the section on simulation.

An idealized RDE is a distributed environment with the following conditions:

- *Closely-coupled relation*: This is the most strong version of close view discussed above. The relation exists in distributed environments if all local *configuration-bits* in active nodes are consistent with the *global configuration-bits* at any time when message deliveries are initiated during the life-time of an environment. The closely-coupled relation makes loosely-coupled distributed systems look like they share a portion of memory, *configuration-bits*, the major feature of closely-coupled distributed systems. This is the reason we call this the closely-coupled relation.
- *Immediate effect*: Immediate effect guarantees that the status of the related nodes remain unchanged during the transmission of messages. This means that messages are received by the destination nodes instantaneously with sending out the messages.

Idealized RDE guarantees 100% reliable delivery of messages.

It is impossible for idealized RDE to exist in a message passing system due to the nature of message passing: transmission time is required. The goal of this paper is not to achieve idealized RDE but instead to build a framework for distributed systems based on an architecture of highly reliable delivery plus configuration exception control; these services are supported in RDE. This framework makes it possible to achieve an almost idealized RDE.

2.3 Distributed Predicting Algorithms For Local Views

A *distributed predicting algorithm* is used in RDE in order to maintain local views. We present three algorithms:

1. Algorithm A: This is the simplest algorithm. Configuration bits are updated in the following cases: i) When a new node comes up, send out control packets "I am up" to all nodes, and then wait for ACK within a timeout slice; this turns on the configuration bits corresponding to the nodes which the ACKs were received from, turning the rest of the configuration bits off. ii) When a sending function is invoked, send data packets to the destination nodes and send control packets to all other nodes, and wait for ACKs from all active nodes within the timeout slice; turn off the configuration bits corresponding to

the nodes from which expected ACKs are not received and turn on the configuration bits corresponding to the nodes from which unexcepted ACKs are received (send out the data packet to these nodes if necessary). iii) When a packet is received, no matter whether it is a data packet or a control packet, send back an ACK; if the configuration bit corresponding to the node that the packet is received from is off, turn it on. It would be unnecessary to send control packets to inactive nodes if the missing rate of packet transmission is equal to zero, where no unexpected ACKs which respond to control packets will possibly come back since all active nodes are noticed when one node comes up.

2. Algorithm B: This algorithm is identical to algorithm A except that the configuration bits is sent out with the packet to active nodes. Each receiving node compares its configuration bits with those received, and then turns off its configuration bits where an inconsistency occurs. The reason for turning off the configuration bit instead of turning it on is because we assume a reliable communication after *k*-retry/timeout is applied. A node changing from inactive to active can broadcast a control packet, but a node changing from active to inactive cannot broadcast any more. The small probability of missing data ensures that almost all inconsistent bits which are on correspond to inactive nodes.

3. Algorithm C: This algorithm is similar to algorithm B except that the node which turns the bits on when eventually receiving unexpected ACKs or turns the bits off when not receiving expected ACKs within the timeout slice sends out its configuration bits to all active nodes after updating. That is, the first node which notices the failure of a node informs all the other active nodes. An active node might receive a notice of the failure of itself due to packets missing in transmission. The node has to immediately broadcast a message "I am still alive!" to correct the inconsistency between local view and global view.

3. Simulation

The purpose of the simulation is to obtain the coupled relation, which indicates the reliability of message passing in a distributed environment based on our architecture. The performance of distributed computing tasks depends heavily on the coupled relation of the environment: distributed computing tasks on a higher coupled relation environment incur lower exception handling costs. Clearly, coupled relation is a measurement of the performance of distributed systems constructed with our model since we require some level of reliability achieved through exception handling. For distributed computing tasks running on a given distributed environment based on different parameters, we adjust the performance to the highest end where any exception handling is incorporated, so the reliabilities of distributed computing tasks are decreasing to the lowest end which is the reliability of the distributed environment. The coupled relation is thus designed to quantify the naked reliability consistent with the nature of the message passing.

3.1 Environmental Factors

The coupled relation is affected by some environmental factors:

1. Distributed predicting algorithms provide different degrees of consistency between local views and the global view. The more precisely distributed predicting algorithms can predict each node's local view, the more reliable the environment.

2. Node failure rate R_{fail} is the number of nodes that fail in a time unit. Since a failed node can not broadcast a message to announce its failure, the node failure rate directly affects the precision of view prediction.

3. Node recovery time is the interval from the time that a node fails to the time it recovers. The reciprocal of node recovery time is node recovery rate R_{rec}.

4. Message arrival rate R_{mesg} is the number of message passing operations in a time unit.

5. Missing rate of data transmission is the probability that data packets are missing or can not be delivered in a certain time slice.

The simulation assumes a distributed computing task running on eight nodes. Each node interacts only through broadcasting messages. Three assumptions are made:

- Poisson arrival rates for node failure, node recovery and message broadcast.
- No partition.
- No duplicate, missing or out of sequence data packets. The missing rate of data transmission is zero. Reliable transmission can be achieved by the mechanisms of *k*-retry/timeout and sequence number.

Three parameters are used in the simulation as follows:

1. Predicting algorithms: algorithm A, B and C as described above.

2. $\rho_m = \frac{R_{rec}}{R_{mesg}}$: the ratio of node recovery rate to message arrival rate

3. $\rho_r = \frac{R_{fail}}{R_{rec}}$: the ratio of node failure rate to node recovery rate

3.2 The Behavior Of The Coupled Relation

Two sets of data were collected for each predicting algorithm, one obtained by fixing ρ_r and changing ρ_m, the other by fixing ρ_m and changing ρ_r. Figure 3 presents the graphs for coupled relation when ρ_r is fixed and ρ_m changes from 0.0 to 2.0. Every curve in figure 3 has two phases. First, as ρ_m increases from zero, the coupled relation decreases dramatically to a minimum point. Second, as ρ_m continues increasing, the coupled relation increases smoothly.

Three things are helpful to understand this two-phase behavior: i) $\frac{1}{\rho_m}$ is the number of messages broadcast during the period of node failure, and every node broadcasting a message can discover the fact of node failure and correct local views; ii) one control message "I am up" is broadcast when a node comes up to end the period of node failure, forcing all local views to be corrected if the data missing rate is zero — the end of node failure period is a check point that guarantees the correctness of local views during the period that no node fails; iii) the coupled relation is important and effective only when data messages are broadcast.

This is the way that the coupled relations were calculated in the simulation: Assume $\frac{1}{\rho_m}$ is N and the number of broadcasting nodes is M. For ease of explaining the two-phase behavior, we pick the case of algorithm C. In the first phase that ρ_m is small (N >> 1), let us focus on the period of node failure since all local views are correct during the period that no node fails. Only the node broadcasting the first message suffers the incorrectness of its local view due to the node failure, and then broadcasts a control message to correct the local views of all other nodes. All nodes that broadcast the following N-1 messages take advantage of correct prediction of local views and N-1 messages are delivered without exception. $\frac{1}{N}$ describes the degree of inconsistency between local views and the global view during the period of node failure. When N increases, the degree of inconsistency decreases. The lower ρ_m indicates higher coupled relation in this phase. In the second phase where ρ_m is not small (N < 1), the effect of the control message "I am up" broadcast when a node recovers from failure is significant, most likely causing the local views of all active nodes to be corrected before any data is broadcast. The interval between two distinct broadcasts is larger than the interval of node failure. The number of control messages "I am up" is more than the number of data messages. The smaller N causes the lower probability of incorrect local views. This results in a high probability of safely delivered messages. The higher ρ_m has the higher coupled relation.

Coupled relations with the different distributed predicting algorithms drop at different rates. This can be explained with N and M as mentioned above, when we focus on the period of node failure. For algorithm A, every node from M suffers the incorrectness of its local view once, and only N-M messages can be delivered without trouble; thus the probability of having incorrect local views (that is, active configuration-bits corresponding to inactive nodes) is $\frac{M}{N}$ and the portion that the environment predicts precisely is $\frac{N-M}{N}$. For algorithm C, only the first node from M suffers incorrect view prediction and the following N-1 messages are delivered without troubles; thus the probability of incorrect local views is $\frac{1}{N}$ and the portion of precise prediction is $\frac{N-1}{N}$. For algorithm B, the number of nodes that suffer the incorrectness is between 1 and M, since the local views of a subset of nodes from M are corrected through comparing the *local configuration-bits* with remote *configuration-bits* in previous message broadcasts. The degree of correctness of local views using algorithm B lies between that of using algorithms A and C. Generally speaking, algorithm C is better than B and algorithm B is better than A when the missing rate is zero.

Figure 4 presents the graphs for coupled relations where ρ_m is fixed and ρ_r changes from 0.0 to 2.0. One phase is shown in most curves (ρ_m is small): when ρ_r increases from zero, the coupled relation decreases. This behavior can be explained as follows:

ρ_r can be considered as the period of node failure, called $T_{inactive}$ if we fix the period that no node fails to 1. During this period, all local views are consistent with the global view under the assumption that the missing rate is zero. All messages sent during this period should be safely delivered if the immediate effect is applied. The probability of incorrect local views due to node failure is $\frac{T_{inactive}}{1+T_{inactive}}$; thus when $T_{inactive}$ increases, the probability increases also. When the probability of suffering incorrectness increases, the coupled relation decreases. By comparing the two sets of graphs, we see that the factor of ρ_m affects the coupled relation more than ρ_r does. Increasing ρ_m has a much more significant effect on decreasing the coupled relation than increasing ρ_r.

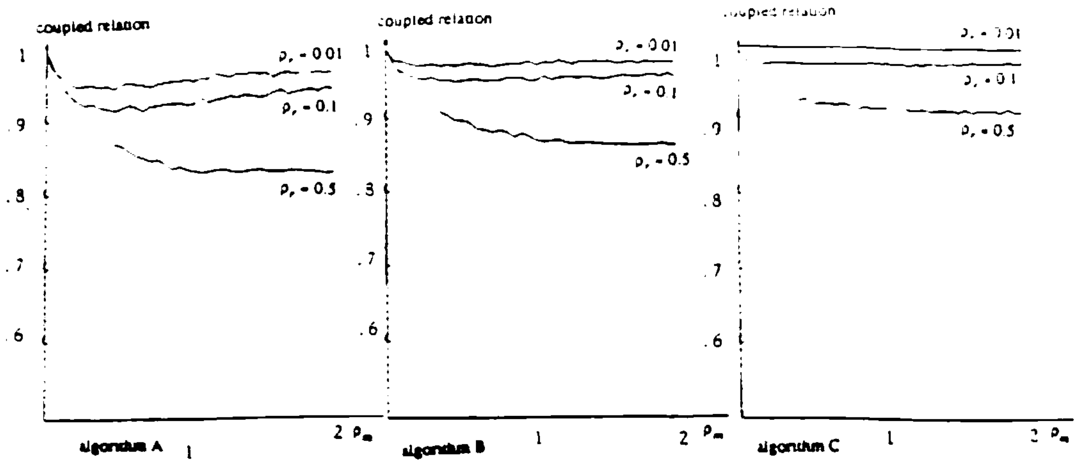


Figure 3: Coupled Relation By Changing ρ_m

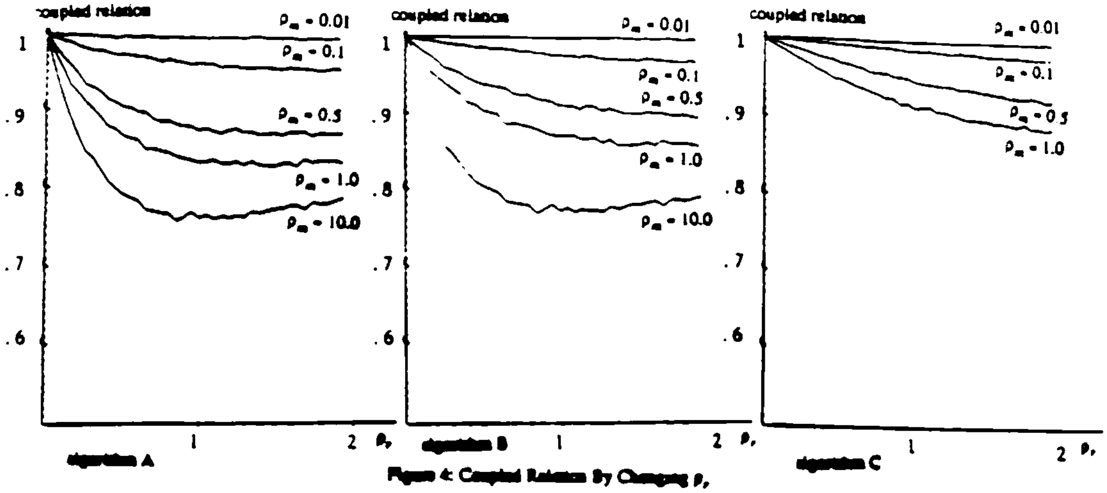


Figure 4: Coupled Relation By Changing ρ_r

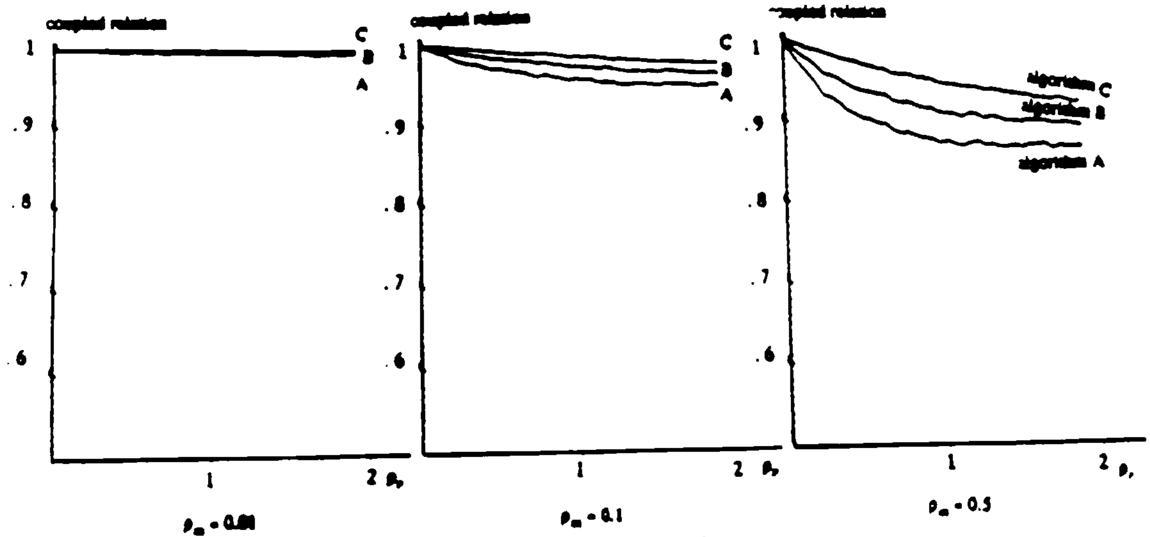


Figure 5: Comparison Of Coupled Relations Between Algorithm A, Algorithm B And Algorithm C

Figure 5 shows the differences among coupled relations based on three algorithms. Algorithm C is clearly better than B, and algorithm B is clearly better than A.

If the assumption of zero missing rate is no longer kept, algorithm C might not be the best because a node might receive wrong information due to missing data, and then it broadcasts the wrong information. For example, a link between $node_1$ and $node_2$ is down for some reason. $Node_1$ broadcasts a message. Clearly, it can not receive any response from $node_2$. Then node 1 assumes $node_2$ is down and broadcasts a message "$node_2$ is down" to all other nodes. Again, $node_2$ cannot receive this message and never has a chance to correct this error. All other nodes update their local views that $node_2$ is down, even though the links between $node_2$ and other nodes are up.

4. View Section for Constructing Distributed Computing Tasks

Our view section model is a programming framework for constructing distributed computing tasks on top of RDE. View sections protect against changes to the imagined shared memory, global view, in the same way that critical sections protect against the change of real shared memory. Certain differences exist: for critical section, shared variables can be locked against being further accessed until they are unlocked. In contrast, the change of global view due to the failure of nodes is totally out of control, so there is no way to prevent the global view from changing. The philosophy is that a view section, which defines a period of time and a sequence of instructions, is declared as a protected section during which the global view is desired to remain the same. If the global view does change during the protected section, a compensation function, defined in the beginning of the view section, is invoked by a notification generated from the underlying RDE. A compensation function behaves as a special form of exception handler. The service supported by the underlying RDE is *configuration exception handling* as mentioned above.

Each view section begins with the statement, *Begin_View_Section*, and may end with the statement *End_View_Section*, or end without the statement *End_View_Section*. We call them close-type view section and open-type view section respectively to distinguish the static characteristics of programming structure. In run time, a view section may end with the statement *End_View_Section* or end in a given time slice. We call them code-bound view section and time-bound view section respectively. An open-type view section must be a time-bound view section, but a close-type view section does not have to be a code-bound view section — it may end with code or with time.

Open-Type View Section

```
vsec:=Begin_View_Section(ngroup,tmslice,cfunc)
....
```

Close-Type View Section

```
vsec:=Begin_View_Section(ngroup,tmslice,cfunc);
....
End_View_Section(vsec);
```

The statement, *Begin_View_Section*, initializes a view section. The first parameter ngroup specifies a set of participant nodes. The second parameter tmslice is the maximum time period in which the tasks executed in the view section are expected to be accomplished. Programmers can set the time slice themselves, or use a time estimation function, provided by the underlying RDE, to get the time slice. Alternatively, they can leave the problem to RDE by setting DEFAULT. The third parameter, cfunc, is an exception handler for when exceptions arise during the view section. Because a computing task may be involved in more than one view section at the same time, vsec acts as a handle that uniquely identifies a particular view section. The handle vsec is very important when nested view sections or overlapped view sections are allowed. For example, an open-type view section can be inside a close-type view section.

The statement, *End_View_Section*, is a function call which ends the view section immediately. If the given time period is consumed before the statement *End_View_Section* is executed, the view section is forced to end by implicitly invoking an RDE procedure which checks the status of the distributed environment and, if necessary, invokes the compensation function. Then the program continues to execute the current statement if the view section is open-type, or jumps to the next statement of *End_View_Section* if close-type.

Another important statement is:

```
Update_View_Section(vsec, tmslice, cfunc)
```

which reinitializes view section vsec and changes the time slice and the compensation function if necessary. If programmers do not want to change the parameters, they can set the parameters to SAME. Parameter ngroup is unnecessary here. This statement can also be used to end a view section if the time slice is set to zero. The possible usage is that it is called from the compensation function to end the view section, when a fatal exception arises.

One typical pattern of view section is

```
vsec:=Begin_View_Section(ngroup,tmslice,cfunc);
multicast(ngroup, message);
for(node i in ngroup is active) receive(message);
End_View_Section(vsec);
```

Nested view sections and overlapped view sections are permitted in our model.

<u>Nested View Section</u>

```
vsec1:=Begin_View_Section(group1,time1,function1);
vsec2:=Begin_View_Section(group2,time2,function2);
End_View_Section(vsec2);
End_View_Section(vsec1);
```

<u>Overlapped View Section</u>

```
vsec1:=Begin_View_Section(group1,time1,function1);
vsec2:=Begin_View_Section(group2,time2,function2);
End_View_Section(vsec1);
End_View_Section(vsec2);
```

Many problems arise due to the time slices, but these are outside the scope of this paper. In the next subsections, we give two examples using view sections.

4.1 Example 1: Summation of Distributed Data

This example does not illustrate the feature of reliability when using view section. It illustrates only how to use view section on top of RDE. Example 2 will describe how reliability is achieved by using the view section model.

A system has N nodes which are each randomly active or inactive. Each node has a variable X that changes in value from time to time. A designated node executes a function that returns the sum of the X values for all active nodes. The multicast function is supported by RDE. Two functions are presented below to solve the problem. One important assumption is that we allow the global view to change during the processing time in order to get the sum from the most recent group of active nodes. The main function is Sum, which broadcasts a message to all active nodes and then waits to receive the X value from each active node.

After it initializes the array X, Sum begins a view section by executing the statement **Begin**_View_Section, which defines a timeout slice, stime, and a compensation function, Check_Sum. Then the Sum function multicasts a request to all active nodes and waits for the X values from all active nodes. It adds each received X value to S until values have been received from all active nodes. Remember the local view might be updated by the compensation function during the processing.

The compensation function, Check_Sum, will be invoked as an exception handler routine whenever the global view is changed. Function Get_Exception_Node will return the node whose status changed to cause the exception. Function Inactive will return true if the current status of node nd is inactive and function Active will return true if the status is active. If the status of the exceptional node nd changes from active to inactive, then it will i) subtract X's value of node nd from the sum ; ii) set X's value of node nd to zero; iii) update view section, which makes the following exceptions happen based on new view, old timeout slice and old compensation function. If the status of the exceptional node (nd) changes from inactive to active, it will i) send a request message to the exceptional node; and ii) update the view section, which makes the following exceptions arise based on the new view, new timeout and old compensation function.

```
Procedure Sum;
begin
 for(node nd in group) X[nd] = 0;
 S := 0;
 vsec:=Begin_View_Section(group,stime,Check_Sum);
 multicast(group,"send back X value");
 for(node nd in group is active)
 begin
  receive(X);
  S := S + X;
  X[nd] = X;
 end;
 End_View_Section(vsec);
 Sum := S;
end;

procedure Check_Sum;
begin
 nd := Get_Exception_Node();
 if(Inactive(nd) = TRUE) then
 begin
  S := S - X[nd];
  X[nd] := 0;
  Update_View_Section(vsec,SAME,SAME);
 end
 else if(Active(nd) = TRUE) then
 begin
  send_to(nd,"send back X value");
  Update_View_Section(vsec,stime,SAME);
 end;
end;
```

4.2 Example 2: Reliable Resource Redistribution

A distributed environment has N+1 nodes, $node_0$, $node_1$, ...,$node_N$, which may be active or inactive. $Node_0$ is the leader which was previously elected by all the nodes. Each node has several resources that might be allocated by a local process. For $node_i$, the set of available resources is R_i. The leader,

$node_0$, invokes a task of resource redistribution upon a request from another node that has consumed all its available resources. The leader broadcasts a message to ask other nodes to relinquish their available resources. After the leader receives all relinquished resources, $R_1+R_2+...+R_N$, it reassigns resources so that the set of available resources for node i is Q_i. Then the leader sends Q_1, Q_2, ..., Q_N to $node_1$, $node_2$, ..., $node_N$, respectively.

The problem is that every node, including the leader, might fail unexpectedly during the process of resource redistribution. We do not want to lose or duplicate resources due to the failure of regular nodes (*i.e.*, not the leader). We also do not want the leader to swallow resources due to the failure of the leader; this might block or dramatically slow down the whole system. We do not address the reassignment problem, but we assume it takes time to complete this task.

The main function is Resource_Redistribution, which divides into three blocks. In the first block, it initializes a view section with compensation function Check_Total. It multicasts a message REQUEST to ask all nodes to give up and send back their available resources R_is, and then waits for all resources to be relinquished. After it receives the available resources from all active nodes, it reinitializes the view section with compensation function Check_Fail and goes to the second block. In the second block, it reassigns resources into Q_i, and sends the Q_i to nodes. One important consideration is that the leader has to ask nodes to lock the resources R_i before they get Q_i back and if a node fails before getting Q_i, it should consider R_i as the available resource when it recovers from failure. This prevents the received resources from premature allocation to local processes until the leader makes sure that the redistributed resources have been safely received by all nodes. If $node_i$ fails during the second block, i) all resources which were already distributed have to be cancelled; ii) the resources received from $node_i$ in the first block have to be discarded so that $node_i$ can consider R_i as its available resources after it recovers from failure; iii) reassignment of resources has to be done again and then the leader redistributes the sets of resources. The view section ends at the end of the second block. In the third block, not inside the view section, it broadcasts a message OK to unlock the resources. Message OK indicates the leader knows all nodes have received the newly assigned resources Q.

Two compensation functions are used in the view section: Check_Total in the first block and Check_Fail in the second block. Check_Total is almost the same as the compensation Check_Sum in the previous example "summation of distributed data" except that resources are used here. In the second block, Check_Fail is the compensation function which, when some $node_i$ fails, asks all the nodes that already received resources Q to give them up, and restarts the second block again. Function Set_Resume is used to jump gracefully to the beginning of the second block, label REASSIGN. We don't care about the case that nodes are restored from previous failures, because it is too late to reassign the resources for the "coming up" node. It has to wait until the next round of resource redistribution.

From the view point of a regular $node_i$, several rules are followed:

1. $Node_i$ considers R_i as its available resources in the following cases:

- $Node_i$ does not receive Q_i within a given time slice, after it sends out R_i. When timeout occurs, the leader is assumed to have failed.
- $Node_i$ fails after it gives up R_i and before it receives its newly assigned resources Q_i.
- $Node_i$ doesn't receive an OK message from the leader within a given time slice. The leader is assumed to have failed. The OK message is sent out by the leader, when the leader makes sure all nodes received newly assigned resources Q_1, Q_2, ..., Q_N.

This is to protect resources from being swallowed by leader, if the leader node fails after it receives part or all of the resources.

2. $Node_i$ considers Q_i as its available resources, if it receives an OK message after it received Q_i from the leader.

3. $Node_i$ considers R_i and Q_i as its possible available resources, if it fails after it receives Q_i and before it receives an OK message. A checking procedure, which checks whether R_i or Q_i is its available resources will be invoked when $node_i$ recovers from failure. That is the procedure: i) $node_i$ broadcasts a message to ask whether Q_i is a valid resource set or not, where we can use version number of Q_i to verify it; ii) if $node_0$, which was the leader when $node_i$ failed, says "no", it uses R_i as its available resources; iii) if any node says "yes", it uses Q_i as its available resources; iv) otherwise, it waits until a node comes up, and repeats the whole procedure. This is to protect resources from been duplicated or lost. A state diagram is presented to describe the behavior of regular nodes.

```
procedure Resource_Redistribution;
begin
  vsec:=Begin_View_Section(group,stime,Check_Total);
  multicast(group,REQUEST);
  for(node nd in group is active)
    receive(R[nd]);
  Update_View_Section(vsec,stime,Check_Fail);
REASSIGN:
  reassign_resources(Q[]);
  for(node nd in group is active)
    send_to(nd, Q[nd]);
  End_View_Section(vsec);
  multicast(group, OK);
end;

procedure Check_Total;
begin
  nd := Get_Exception_Node();
  if(Inactive(nd) = TRUE) then
    Update_View_Section(vsec,SAME,SAME);
  else if(Active(nd) = TRUE) then
  begin
    send_to(nd, REQUEST);
    Update_View_Section(vsec,stime,SAME);
  end;
end;

procedure Check_Fail;
begin
  nd := Get_Exception_Node();
  if(Inactive(nd) = TRUE) then
  begin
    for(node nd to which Q[nd] has been sent)
      multicast(group,DISCARD);
    Update_View_Section(vsec,SAME,SAME);
    Set_Resume(REASSIGN);
  end;
end;
```

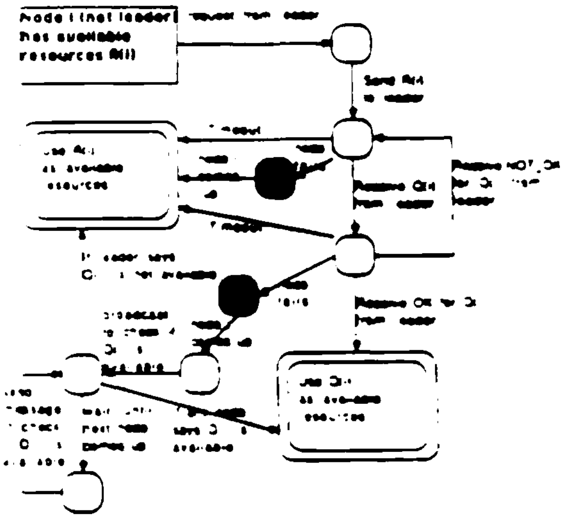


Figure 2. State Diagram For A Regular Node (Not Leader) In Example "Reliable Resource Redistribution"

5. Related Work

Peterson's *conversations*[11] is another IPC abstraction that attempts to achieve reliability in distributed computing by incorporating a notion of *view*. In a conversation, however, a view is a *context graph* giving the partial ordering of all past messages. As with our RDE, the global view is the truth and the local view is what distributed nodes know about the truth. Distributed nodes sense the truth through message passing, and the difference between the global view and a local view is due to the unreliability of message passing. That is, a node may not be aware of the entire context due to missing or out of order messages, as well as node failures or network partitions. Each message is passed with the entire context graph known by the sender, so the receiver has the opportunity to update its own and/or the sender's view of the truth with any messages included in one but not the other.

Peterson's conversations and our RDE have the following features in common:

- Global view: There exists a global entity, the truth, which might change from time to time. In the conversation IPC abstraction, it is the context graph. In our model, it is *global configuration-bits*.
- Local view: Every node has a view that represents its knowledge about the global view. Local views are updated incrementally through message passing and are rebuilt after node failure or network partition. In conversations, it is participant *p*'s view of the context graph. In our model, it is *local configuration-bits*.
- Knowledge basis: Distributed nodes behave according to their local views, and local views and operations are mutually affected. In conversations, the operations are the standard send() and receive(). Our model also provides sendto() and multicast().

Peterson attempts to achieve *ordering of messages* for group communication, whereas we try to attain *safe delivery of messages* for group communication. This difference in goals explains the differences between views. The dynamic views used in conversations preserve context information, while the (conceptually) static views used in RDE preserve configuration information. Conversations support ordered broadcast while RDE protects against node failure.

Thus, conversations and our model are really complementary. It would be nice to implement conversations on top of RDE, using view sections to build the component of conversations that protects against node failure. This would solve certain problems of conversations, such as nodes expecting an acknowledge from a failed process and lack of reliable broadcasts. It would also be easy to use view sections to rebuild the context graph when the system recovers from node failure or network partition.

6. Conclusion

We propose a reliable distributed environment (RDE) among large groups of nodes to ensure the reliability of complicated communication patterns as virtual circuits are already applied to pairs of nodes to ensure the reliability of simple communications. RDE serves as a basic communication environment to handle large scale and/or intelligent distributed systems. RDE provides reliable communication services and configuration services which do not exist in traditional communication environments.

We characterize the reliability of a distributed environment by its *coupled relation*, which is based on different predicting algorithms, ρ_m and ρ_r. Our simulation results clearly illustrate how the coupled relation affects the performance of a distributed environment.

We have demonstrated the utility of our *view section model* by our reliable resource redistribution example. We believe this model, which supports a high level of abstraction for handling low level environmental changes, will prove to be a good programming framework for constructing reliable distributed computing tasks.

We expect our network architecture, with view section as the top layer, RDE in the middle and datagram communication on the bottom, to become increasingly important due to the movement towards large scale distributed systems and intelligent distributed systems.

Acknowledgements

Mr. Hseush is supported in part by the New York State Center for Advanced Technology — Computer Information Systems Research. Dr. Kaiser is supported in part by grants from AT&T Foundation and Siemens Research and Technology Laboratories, and in part by a DEC Faculty Award.

References

1. K. Ravindran & Samuel T. Chanson, "State Inconsistency Issues In Local Area Network-based Distributed Kernels".

2. W.L. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept", *Software — Practice & Experience*,Vol. 11No. 5May 1981, pp. 435-466.

3. Gregory R. Andrews and Fred B. Schneider, "Concepts and Notations for Concurrent Programming".

4. J. Eliot B. Moss, "Nested transactions: An approach to reliable distributed computing", Tech. report, Massachusetts Institute of Technology, April 1981.

5. Butler W. Lampson, *Atomic Transactions*, Springer-Verlag, 1981, ch. 11.

6. H.S. Stone, *Introduction to Computer Architecture*, SRA, 1980.

7. Andrew S. Tanenbaum, *Computer Networks*, Prentice-Hall, Inc., Vrije University, Amsterdam, The Netherlands, 1976.

8. Jeff Kramer and Jeff Magee, "Dynamic Configuration For Distributed Systems", *IEEE Transactions on Software Engineering*,Vol. SE-11No. 4April 1985, pp. 424-434.

9. Stuart Sechrest, "An Introductory 4.3 BSD Interprocess Communication Tutorial", Tech. report, Computer Science Research Group, Computer Science Division Department of Electrical Engineering and Computer Science University Of California, Berkeley, 1986.

10. Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, "An Advanced 4.3 BSD Interprocess Communication Tutorial", Tech. report, Computer Science Research Group, Computer Science Division Department of Electrical Engineering and Computer Science University Of California, Berkeley, 1986.

11. Larry L. Peterson, "Preserving Context Information in an IPC Abstraction", *Proceedings of Sixth Symposium on Reliability in Distributed Software and Database Systems*, 1987.

Software Development Environments for Very Large Software Systems

Gail E. Kaiser, Columbia University, Department of Computer Science, New York, NY 10027

Yoelle S. Maarek, Technion, Israel Institute of Technology, Computer Science Department, Haifa, 32000 ISRAEL

Dewayne E. Perry, AT&T Bell Laboratories, Computer Systems Research Lab, Murray Hill, NJ 07974

Robert W. Schwanke, Siemens Research and Technology Laboratories, Princeton Forrestal Center, Princeton, NJ 08540

December 1987

CUCS-279-87

Abstract

This technical report consists of the three related papers. *Living with Inconsistency in Large Systems* describes CONMAN, an environment that identifies and tracks version inconsistencies, permitting debugging and testing to proceed even though the executable image contains certain non-fatal inconsistencies. The next two papers are both from the INFUSE project. *Change Management for Very Large Software Systems* presents the new non-Euclidean hierarchical clustering algorithm used by the INFUSE change management system to cluster modules according to the strengths of their interdependencies. *Models of Software Development Environments* presents a general model of software development environments consisting of three components — policies, mechanisms and structures — and classifies existing and proposed environments into the individual, family, city and state classes according to the size of projects that could be adequately supported.

Prof. Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. When this research was conducted, Ms. Maarek was a visiting PhD student at Columbia University.

Final version to appear in International Workshop on Software Version and Configuration Control, Grassau, West Germany, January 1988.

Living With Inconsistency in Large Systems

Robert W. Schwanke
Siemens Research and Technology Laboratories
Princeton, NJ 08540

Gail E. Kaiser[*]
Columbia University
Department of Computer Science
New York, NY 10027

31 August 1987

[*] Supported in part by grants from Siemens Research and Technology Laboratories, AT&T Foundation, and New York State Center of Advanced Technology - Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

Abstract

Programmers generally want to be sure that the systems they are building are *consistent*, both with respect to source code versions used, and with respect to type safety. Most modern high-level language systems enforce this consistency upon the system instances they build. However, in a large system this can lead to very large recompilation costs after small changes. Therefore, programmers often circumvent enforcement mechanisms in order to get their jobs done. The CONMAN *con*figuration *man*agement project explores the premise that some degree of inconsistency is inevitable in software object bases, and that programming tools should be designed to analyze and accomodate it, rather than to abhor it. The CONMAN programming environment will help the programmer contend with inconsistency by automatically identifying and tracking six distinct kinds of inconsistencies, *without* requiring that they be removed; by reducing the cost of restoring type safety after a change, through a technique called *smarter recompilation*; and by supplying the debugger and testing tools with inconsistency information, so that they can protect the programmer from flaws in the code.

1. Introduction

Every programmer remembers wasting large amounts of time looking for a bug caused by changing and recompiling one source file and failing to recompile a related file. This kind of problem has made the Unix$^{\text{TM}}$ *make* tool [3] very popular; when invoked after a change to a source file, *make* rebuilds every file derived (directly or indirectly) from the changed file.

Programmers generally want to ensure that the systems they are building are *consistent*. For example, they want to know that the object code they are running was built from the exact source code they are looking at, rather than from some previous version of the source code. They also want to ensure that the executable program is *type safe*; that is, that it satisfies the type rules of the programming language. Most modern high-level language systems enforce this consistency upon the system instances they build. In a large system, however, this can lead to very large recompilation costs even after small changes. Therefore, programmers often circumvent enforcement mechanisms in order to get their jobs done.

This practice is not only commonplace; it is commendable! The programmer can do it successfully by using design knowledge to decide which inconsistencies are harmless and which are dangerous. Allowing inconsistency can speed up the edit-compile-debug cycle, and can also reduce the coordination needed between programmers. Both benefits improve productivity dramatically.

The CONMAN *con*figuration *man*agement project is exploring the premise that some degree of inconsistency is inevitable in software databases, and that programming tools should be designed

to analyze and accomodate it, rather than to abhor it. The CONMAN programming environment helps the programmer contend with inconsistency by:

- Automatically identifying and tracking inconsistencies: CONMAN classifies each inconsistency into one of six categories, and tracks it for the programmer, *without* requiring her to remove it right away.

- Reducing the cost of type safety: CONMAN's type safety is based on a constraint called *link consistency*, which is less stringent than in conventional systems. This permits use of a technique called *smarter recompilation* to reduce the cost of restoring type safety after a change [15].

- Supporting debugging and testing: The debugger automatically stops execution upon reaching inconsistent code, thus helping to prevent crashes. The test coverage analyzer tells the programmer which tests can be executed in <the presence of an inconsistency.

This paper begins by presenting several scenarios in which allowing inconsistency is more cost-effective than removing it. Then it describes the six kinds of consistency that CONMAN recognizes automatically. Next, it explains how smarter recompilation uses link consistency to decide which modules really must be recompiled after a source code change. Finally, it describes how the CONMAN programming environment uses consistency analysis to help the programmer build, debug and test inconsistent systems.

2. Beneficial Inconsistency

Inconsistency is commonplace in software project libraries. A project library typically contains many system configurations, where each configuration might contain requirements, specifications, code, test data and documentation. Informally, a project library is inconsistent if it contains direct contradictions. For example, if a global data type is somehow defined differently in different parts of a configuration, this constitutes a contradiction (because most languages permit only one definition of each global identifier). On the other hand, two distinct system configurations may define the type differently, and that would not be a contradiction.

Inconsistency is likely to occur when permitting it is more cost effective than forbidding it. For example:

- Debugging and testing under deadline pressure. On fixing a bug, the programmer should recompile the minimum amount of code necessary to continue testing. She can wait to recompile the rest of the system until she goes home for the night.

- Debugging an incomplete implementation. In a language such as Ada$^{(R)}$, with specifications separated from package bodies, an early version of a package body might not contain all of the procedures. The programmer should not be distracted from her creative task by the tedium of writing stubs. (Wolf studies this form of incompleteness [18].)

- Changing requirements after implementation is under way. When requirements change, it may be easier to start by combining the new requirements with the old implementation -- even though they contradict each other -- rather than keeping them in separate system configurations until they agree.

- Handling "software rot". Sometimes a bug fix introduces new bugs. Until the new bugs are resolved, debugging may be easier if some parts of the system use the old version of the code, while others use the new version.

- Large teams debugging related changes. During large system maintenance, a single change request often involves several modules and the interfaces between them. Each team member would debug her changes independently, before integrating them with the work of others. To do so she should build an executable system instance with whatever versions of others' modules she deems appropriate, even if some of them still use obsolete, incompatible interface specifications.

This last example, when elaborated, provides many clues as to how a programming environment should support programming with inconsistency. Consider a typical operating system maintenance project, having [5]

- 1,000,000 lines of source code,

- 300 programmers,

- a new release about once per year,

- 300,000 lines of new or changed code per release,

Suppose there were one bug for every 30 lines of changed code, the syntax is correct but before any debugging or testing. That would add up to about 10,000 bugs per release.

Many module changes include modified interfaces. Suppose that each programmer has been assigned to modify a different module. Because tasks progress at different rates, and because some tasks must be redone, several new versions of each module will be produced. Each programmer is responsible for debugging and testing her own code as well as she can before releasing it to others. To do so, she selects the versions of other modules that she thinks will work best with her module. However, the ones she wants to use may not be ready yet. She might choose not to simulate them with a test harness, because test harnesses are often too expensive for early debugging and unit testing. They must be updated whenever the interface changes, which requires both manpower and calendar time. Therefore, programmers often build inconsistent configurations of the real system to use for debugging. In fact, large projects often assign their best analysts to figure out workable, albeit inconsistent, configurations for debugging and testing.

To build, debug and test inconsistent systems, programmers need tools that

- Identify and evaluate the severity of inconsistencies.
- Display the inconsistency information in a useful way, such as by incorporating it in a browser or by using it to compare several alternative module versions, none of which is completely compatible with the rest of the system.
- Protect the programmer from system crashes due to known inconsistencies, by placing firewalls around dangerous code.

3. Kinds of Consistency

CONMAN formalizes the concept of inconsistency by defining six distinct kinds of consistency, to use for classifying inconsistencies it discovers in programs.

We use the term ***system instance*** to mean an executable representation of a program, typically created by compiling numerous separate program units and linking them together. We assume that the programming language specifies some form of static type checking, and that the programming environment provides a way of uniquely identifying versions of both source code files and derived files (such as object code files). The six kinds of consistency are:

- *Full consistency*: A system instance satisfies the rules that the programming language specifies for legal programs, insofar as they can be checked prior to execution. It also must be version consistent, as defined below.

- *Type consistency.* The system instance satisfies the static type checking rules of the programming language.

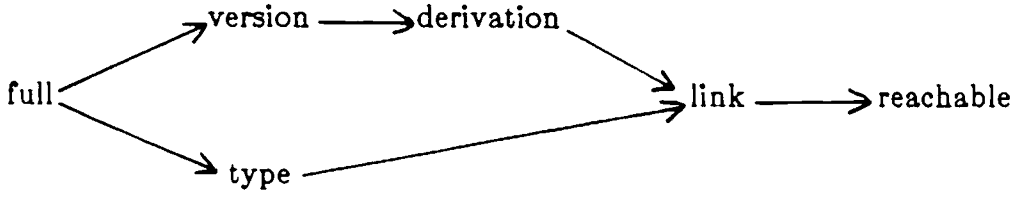
- *Version consistency.* The system instance is built using exactly one version of each logical source code file.

- *Derivation consistency.* The system instance is operationally equivalent to some version consistent system instance (which need not have actually been built).

- *Link consistency.* Each compilation unit is free of static type errors, and each symbolic reference between compilation units is type safe according to the rules of the programming language.

- *Reachable consistency.* All code and data that could be accessed or executed by invoking the system through one of its entry points are type safe.

The definitions above have the following partial ordering:



3.1. Full Consistency

The strongest form of consistency is full consistency. The definition tries to capture the ideal world. For example, a system written in Ada is consistent when it is built with exactly one version of each compilation unit, and the units have all been compiled without error in an order compatible with the inter-package dependencies, and then linked.

3.2. Type Consistency

Type consistency depends only on those language rules that deal with the types of identifiers. Operationally, a system instance is type consistent if the compiler reports no type errors for any separately compiled component, and if each identifier whose scope spans more than one compilation unit has the same type in every such unit. (For the C language, the rules checked by the Unix *lint* tool [6] define type consistency across boundaries of separately-compiled modules.)

3.3. Version Consistency

Version consistency is the system property enforced by Unix *make*. For example, if a system written in C contains a source file named "symtab.h", then *make* ensures that all files that `include` it (incorporate its text) are compiled with the latest version.

Version consistency is also important because it provides a practical means of ensuring (or circumventing!) type consistency. Many language systems implement type checking across separately compiled modules by using a file of definitions, called an "include file", to define the types of the identifiers exported from a compilation unit. If the same version of the include file is used to compile the exporting module and every importing module, then the exported identifiers will have the same type throughout their scopes. Conversely, one can trick a compiler into generating code for a module that is not type consistent with other modules, by using different versions of the include file when compiling different modules.

The definition of version consistency includes the word "logical" to cover a special class of systems in which two or more versions of a module are included by design. For example, a test configuration might be created to compare the behavior of two versions of a module. Its system construction model (cf. DSEE [9], Cedar [8]) would treat the two versions as separate logical entities during compilation and linking. A version consistent instance of this system could still use two different versions of the module, because the versions would implement two different logical modules.

3.4. Derivation Consistency

Derivation consistency includes the class of systems that one can build by foregoing unnecessary recompilations, and then use as if they were version consistent. For example, when a type is changed in an include file, only the modules that use the changed type need to be recompiled. Other modules that include the changed file, but do not use the type that was changed, need not be recompiled. Linking the object modules together produces a system that is equivalent to one where all modules were recompiled to use the new version of the changed include file.

3.5. Link Consistency

Link consistency is weaker than type consistency, because it enforces type safety pairwise between compilation units, rather than requiring types to be defined and used consistently system-wide. Nonetheless, this definition is sufficient to support debugging, because the actual executable code is all type safe according to the rules of the language. If each object module is internally type safe, and every data path between modules is type safe, then there is no place in the system where machine code that expects data of one type can operate on data of some other type.

Link consistency can be achieved without type consistency by using different versions of include files with different compilation units. Two units need to use equivalent versions of an included definition only if the link-time interface between them is affected (directly or indirectly) by that definition.

Link consistency describes some situations where a widely-used definition has been changed, but only some of the places where it is used have been rewritten to accomodate the change. Consider a system in which one module defines the type **linked list**, and two other subsystems each use **linked lists** internally, but do not pass **linked lists** between subsystems. This example is depicted in figure 3-1.

Suppose it is decided to change the implementation from singly-linked lists to doubly-linked lists, to enable sequencing in both directions. The programmer would like to try out the doubly-linked implementation in a limited context, before rewriting all of the places it is used. If she rewrites and recompiles the **linked list** module and just one of the subsystems that uses it, the system instance will be link consistent (because every module and every link is type safe), but not type consistent (because some modules were compiled with the singly-linked implementation, and some with the doubly-linked implementation). Assuming that the list representation is directly manipulated by the subsystems that use it (to increase efficiency), the programmer cannot compile the

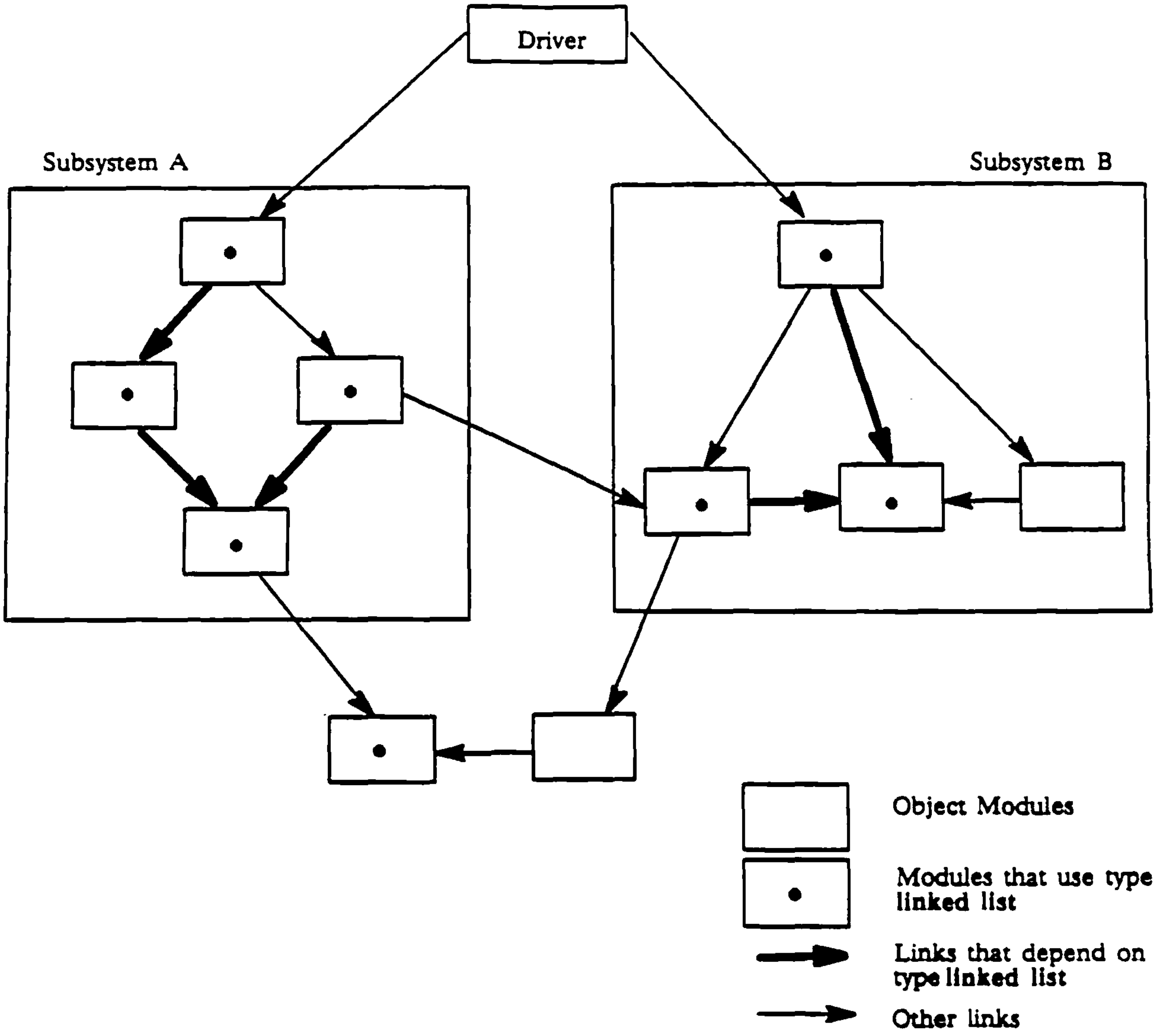

Driver
Subsystem A
Subsystem B
Object Modules
Modules that use type linked list
Links that depend on type linked list
Other links

Figure 3-1: Clusters That Use a Type Independently

second subsystem with the doubly-linked implementation until she rewrites it. Recompiling without rewriting would give lots of error messages, and probably no object code.

Such independent uses of a global type are consistent with sound design principles. A large system is frequently layered into levels, where each level uses services provided by the levels below it, and provides services to the levels above it. In a system that provides a broad range of end-user services, it is not unusual for the middle layers of the system to contain several subsystems that do not call each other at all. In that situation a service type defined by a lower level could be used independently by the subsystems at the next level.

Besides global types, several other language constructs permit multiple coexisting definitions without sacrificing link consistency. For example, Ada's inline procedures and generics both cause a definition to be instantiated separately at each place where it is used. Usually, separate instances of a generic package are treated as unrelated at run time, even though they were derived from a common definition. (Of course, Ada's rules currently forbid version inconsistency.)

3.6. Reachable Consistency

Reachable consistency is useful during development when service routines are written before the external interfaces that use them are ready. Any type errors in unused routines can not interfere with debugging the code that is reachable.

3.7. Automatic Checking

CONMAN checks all six kinds of consistency automatically. Version consistency is checked by straightforward configuration management methods. Type consistency and derivation consistency are checked by the methods used in *smart recompilation* [17]. (Full consistency simply means version consistency and no compilation errors.) Link consistency is checked by a simple method described in the next section. Reachability is checked by incremental, interprocedural data flow analysis, recently made efficient by Ryder and Carroll [14].

4. Reducing the Cost of Consistency

The Unix *make* tool restores version consistency by rederiving any output files that are older than the current versions of the input files from which they are supposed to be built. This can cause many recompilations after only a small change.

Toolpack [12] and smart recompilation reduce the cost of restoring consistency by restoring only *derivation* consistency. Both systems maintain a single, consistent version list of the "latest versions" of each file. They reduce recompilation costs by not rederiving a file when the existing derived file is operationally equivalent to what would be created by rederiving it with the new source file versions.

Toolpack defines "operationally equivalent" to mean "identical contents"; it permits certain attributes such as timestamps to be different. Toolpack uses the same "older than" rule as *make* to trigger recompilation, but avoids some processing steps by noticing when a certain step produces an output file with contents identical to the one it is replacing. This means that using the new output file in a subsequent translation step would be equivalent to using the old version, so the next step is avoided unless other inputs have changed.

Smart recompilation determines equivalence by extracting, from the inputs to a compilation, the set of declarations that actually affect the output files; two output files are equivalent if they are derived from equivalent extracted inputs. (The output files are also allowed to include unused code that differs.) Smart recompilation preprocesses each changed file to identify the declarations that have changed in it. The method then recompiles only the files that actually contain or use the changed declarations.

Smart recompilation succeeds because it performs only local semantic analysis, which it can do cheaply. Local semantic analysis examines each source file in isolation. Any identifiers occurring free in that file are assumed to be declared in some compatible way; they are typically bound by include statements to other files. The analysis produces a dependency file listing the identifiers exported by that file, and the free identifiers on which they depend. The details of smart recompilation are thoroughly explained in [17].

4.1. Checking Link Consistency

To simplify the following sections, we limit our discussion to a simple Pascal programming system, such as provided by the Berkeley Pascal compiler running on Berkeley Unix 4.2. This environment provides a version of Pascal that has been augmented with a separate compilation facility. Procedure headers can be separated from procedure bodies. Typically, the interface to a module is placed in a separate "include" file, which is included in the module that provides the interface and in every module that uses the interface. In the remainder of this paper, we use the term "module" to refer to a normal compilation unit, and "file" to refer to a module or an include file. Our discussion does not cover overloading nor identifiers that are moved between modules during a change. These extensions can be handled analogously to the way smart recompilation handles them.

Link consistency is defined on links between object modules. A link is a (definition, use) pair consisting of an identifier declared *global* in the object module that defines it, and *external* in the object module that uses it. A link is consistent if the definition and the use were compiled using equivalent declarations of the identifier's type. For example, if a procedure **P** with one parameter of type **T** is exported by one module and imported by another, then the two modules must agree that **P** has only one parameter, that its type is **T**, and that **T's** type is equivalent in both modules.

To check link consistency, we first identify the source code constructs that produce global and external references. Then, we use preprocessing methods derived from smart recompilation to analyze dependencies involving these constructs.

The only two kinds of object module links in Pascal are variables and procedures. Where Pascal programs define enumerations, records, constants, etc., the compiler translates them directly into object code, without leaving any links to external identifiers. We know, therefore, that a link exists only where a procedure or variable is exported from one module and imported by another.

To check link consistency, we augment the smart recompilation preprocessor in two ways:

- We divide dependencies into *interface* dependencies and *implementation* dependencies. For example,

  ```
  extern
  procedure P(a:T);
          var b:V;
              . . .
  ```

  This procedure has an interface dependency on type **T**, and an implementation dependency on type **V**.

- For each exported procedure and variable, we record its type signature, in which bound type names are replaced by their definitions, but free type names are treated as primitive. For example,

  ```
  (import type R)
  type Q is integer;
  type T is record
          a: Q;
          b: R
          end

  extern var v: T;
  ```

  In this case, v's type signature would be **record(integer,R)**. (This kind of type signature defines type safety by structural equivalence. It can be easily modified to use name equivalence instead.)

To test whether a link is consistent, we compare the versions of the identifiers that affect the definition site and the use site. We do so in the following steps:

1. Determine which source file versions to associate with the definition site, and which to associate with the use site. These can either be the files that were actually used, or files that are proposed to be used.

2. For both the definition and use sites, locate the source file version that defines the identifier's type.

3. Compare the two definitions for equivalence, as follows:

   a. If the version numbers are different, compare the type signatures. If they are different, the definitions are not equivalent.

   b. For each free identifier in the type signature, compare its two definitions (in the "definition site" versions and the "use site" versions) for equivalence, using this same algorithm recursively.

c. If all the free identifiers in the type signature are equivalent. the definitions are equivalent.

4. (The results of every comparison should be saved for re-use should the type appear again elsewhere in the signature, or in the signature of another link between the same pair of modules.)

4.2. Smarter Recompilation

Smarter recompilation works by finding clusters of modules that must agree on certain identifier definitions in order to be link consistent. Specifically, clusters are defined with respect to a specific set of global identifiers. Two modules are in the same cluster if and only if they are connected by a link that depends on any of those identifiers. (Modules whose interfaces don't depend on the identifiers at all are not placed in any cluster.) Smarter recompilation saves processing time and programming time whenever a system contains two or more clusters with respect to a set of changed identifiers. The method reduces to smart recompilation when this definition causes all modules to be in the same cluster. It starts with the files that have changed, and at least one module that must be recompiled to test the changes. It then "grows" a cluster of modules that are transitively connected to the starting module via links affected by the changes. These are the other modules that must be recompiled. The algorithm proceeds as follows:

1. Begin with a previous system instance. all relevant source file versions, and the results of preprocessing each of the source files. These results are collected in a data structure that indexes all links, so that it is easy to find which links to check when deciding to recompile a module. The data structure is updated incrementally each time the system instance is modified.

2. Ask the programmer to select a set of file versions she wishes to debug or test. There can be at most one version of each logical module in the system, but the programmer need not choose versions of modules she does not care about.

3. Use smart recompilation to select a set of *build candidates*. Smart recompilation requires there to be a set of "new" file versions and a set of "old" file versions. For this purpose, the versions chosen by the programmer are the new ones, and any conflicting versions are the old ones.

4. Ask the programmer to select an initial *build set* from the candidates. These modules define the context in which she wants to debug or test her change.

5. For each new member of the build set,

    a. Determine which versions of the source files will be included when it is recompiled. Use heuristics to select versions that the user left unspecified, such as "latest", "whatever was used before", or "whatever has already been used in the build set".

    b. If the module's source code has changed, update the link index to reflect any changes.

    c. Using the proposed version bindings, check the consistency of each link between the new member and other modules.

    d. Augment the build set with any candidates that have become link-inconsistent with it.

The total time to check consistency is proportional to $B * I * T$, where B is the size of the build set, I is the average number of identifiers imported and exported from a module, and T is the average number of identifiers that must be tested for equivalence in the course of validating a link.

Smarter recompilation can be generalized to more complicated translation tools, and additional kinds of derived files. For example, consider a system written in Ada. The Ada compiler would generate interface files (`.int` files, containing compiled specifications) and object code files (`.obj` files, containing package bodies); the compiler would read in interface files when compiling modules that depended on them. Suppose main subprogram **X** depends on package specifications **Y** and **Z**, and package specification **Y** depends on **Z**. Compiling **X** requires a consistency check between **Y** and **Z**, to ensure that **Y** was compiled with a compatible version of **Z**. This processing model is diagrammed in figure 4-1.

In this situation, the concept of "link" generalizes to "name binding". Each compilation step resolves free names in some of its inputs by binding them to definitions exported by other inputs. Since any exported definition could be involved in a binding,

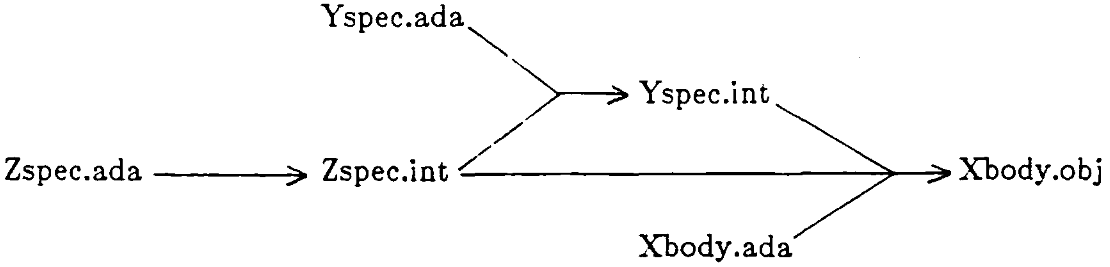


Figure 4-1: Compiling a Small Ada Program With Transitive Dependencies

the preprocessor would keep type signatures for all exported identifiers. Because the inputs to a compilation step are sometimes produced by other compilation steps, there can be version conflicts between inputs to compiles as well as to the link step. The consistency checking algorithm must be augmented to account for such complications in the version selection lists.

Smarter recompilation can be generalized further, to a broad class of translators and derived files, including program generators (such as Unix utilities *lex* and *yacc*), and distributed execution environments. "Compilation" generalizes to any translation step that produces an identifier definition or use based on input definitions and uses. For each "source code" language in the system, one would look for the kinds of identifier declarations that translate into unresolved references in derived files. For each such kind of identifier, a preprocessor would perform local semantic analysis to determine the equivalent of a type signature. Then, each tool that performs name binding can be preceded by an analysis step that uses version lists and type signatures to identify link inconsistencies.

In summary, smarter recompilation reduces the cost of restoring consistency by enforcing only link consistency, rather than derivation consistency. It interacts with the programmer to choose versions relevant to the current task, then performs the least number of compilations necessary to construct a system instance that is link-consistent with those choices.

5. An Environment for Programming with Inconsistency

CONMAN is a programming environment that helps the programmer interactively construct and debug inconsistent systems. The systems may contain different kinds of inconsistency in different places. The environment consists of an object base and a set of tools, consisting of a browser, a compiler, consistency analyzers, an incremental linker, a flow analyzer, a debugger, a test coverage analyzer, and an automated maintainer's assistant. Each is based on available technology, modified to handle inconsistent systems.

The object base is an integrated database of software artifacts [11, 1]. Each file is stored as an object, together with attributes and relations that represent its relationships to other parts of the system. The objects belong to a class hierarchy, with multiple inheritance. Tools in the system can be classified as either foreign tools or native tools. Foreign tools have no knowledge of the environment; they exchange data with the environment through an envelope that sets up an execution environment, calls the tool, and collects its results. Native tools can use the object base directly, such as to store dependencies between source files or to analyze inconsistencies in a desired system instance.

The compiler and linker are augmented with preprocessors to collect type signatures, which the analyzers then use to detect inconsistencies.

The browser helps the programmer construct a description to build. (We call this description a BCT for compatibility with the Domain Software Engineering Environment's (DSEE's) Bound Configuration Thread [9].) A structure editor is a promising type of browser for this application. Through it, the programmer can not only construct the BCT itself, but can also examine its connections with the rest of the object base.

The programmer starts by examining the BCT for some previous system build. The editor presents her with all the new module versions that have been created since the last system build, and asks her which ones she would like to use. The programmer assigns new version bindings to the derived objects she wants rebuilt. As the programmer makes the version choices, the editor highlights version inconsistencies and schedules background tasks to classify them further. Zooming shows details of an inconsistency, including its severity and the specific identifiers involved. The programmer can respond to an inconsistency by:

- Selecting modules to recompile.
- Choosing different source versions.
- Substituting previously compiled object files from the derived object pool (cf. DSEE).
- Approving the inconsistency.

As each part of the BCT is approved, its derivation begins. Any warning or error messages that result are presented to the programmer, who can further modify the BCT if she wishes.

The linker and debugger cooperate to protect the programmer from link inconsistencies. The linker inserts a debugger hook at each inconsistent link, so that execution will stop before the code that uses the link is executed. The debugger then permits the programmer to either move the program counter to a safer place, or continue execution at her own risk.

The BCT description language allows the programmer to permit two versions of an object module to coexist. The linker supports this by accepting multiple definitions of global identifiers, and linking each use to the definition with the correct type.

The test coverage analyzer produces a database for each test indicating the code it covers. On request, it compares this data to the link inconsistencies in a system instance, and tells the programmer which tests are safe and which are not.

The maintainer's assistant is facility for automating mundane programming tasks in a controlled way, called *opportunistic processing*. Whenever a programmer makes a manual change to a source file, it schedules appropriate analysis and compilation tools to run in background, as resources permit. It monitors the costs of compilation and linking, and uses them to estimate the costs of rebuilding after a change. This information is fed back to the user through the browser. The analyzer performs the consistency analysis in background, so that the information is ready when the programmer is ready to edit her BCT. It also maintains an agenda of modules needing rewriting due to changed interfaces.

This combination of tools helps the programmer keep track of inconsistencies, analyze their severity, estimate the cost of recompiling to remove them, and helps select test cases that avoid them. It also protects the programmer from inadvertently executing inconsistent code, while still allowing her to do so if she insists.

6. Implementation

Smarter recompilation has been implemented for the C language, as a Master's thesis at Columbia University [10]. It was constructed by making source code modifications to the portable C compiler and *make*. The prototype successfully handles such details as macros, structs, unions, and even bit field sizes and anonymous struct fields. Although it has not been tested on large systems, it demonstrates that the cost of adding the functionality to existing tools is reasonable.

The CONMAN programming environment is being assembled from a collection of other systems being developed and/or used at Siemens RTL. The object base and controlled automation system are being designed in conjunction with the Marvel project [7]. The browser is being implemented with the DOSE structure editor prototyping system [2]. The system modeling language draws ideas from both DSEE and Cedar, but adds facilities for conveniently naming and manipulating derived objects, and for mapping source-language dependencies into build step input-output dependencies. For example, a system model could declare that one source file called procedures in another source file; the system builder would automatically link the second file into system instances that used the first. The debugger will be the Sun Unix *dbxtool* [16], which will be primed with a set of breakpoint commands generated by the linker. Test coverage tools and methods will be drawn from the Asset project [13, 4]. Reachability analysis will be based on Ryder's methods, in a future version of the system.

7. Conclusions

Inconsistency is commonplace in real software projects. It is permitted to remain because it is often more cost-effective than consistency.

Automatically recognizing several gradations of consistency permits the programmer to choose the level appropriate to her task. Better tools can reduce the cost of restoring consistency, but not the cost of rewriting all the code affected by a change. Smarter recompilation permits derivation inconsistency without sacrificing run-time type safety, and thereby permits some rewriting to be deferred, reducing the length of the edit-compile-debug cycle and reducing the amount of synchronization needed between programmers.

The CONMAN configuration management project is developing a programming environment that helps a programmer to select different degrees of consistency in different parts of her system. The tools will recognize and keep track of inconsistencies for her, and place firewalls around them during debugging, but will not force her to remove them. By this approach, CONMAN will help the programmer live with inconsistency.

8. Acknowledgement

Our thanks to Walter Tichy for his many helpful comments on earlier versions of this paper, and to Harris Morgenstern for implementing smarter recompilation as his master's project at Columbia University.

References

[1] Philip A. Bernstein.
Database System Support for Software Engineering.
In *9th International Conference on Software Engineering*, pages 166-178. Monterey, CA, March, 1987.

[2] Peter H. Feiler, Fahimeh Jalili, and Johann H. Schlichter.
An Interactive Prototyping Environment for Language Design.
In *Proceedings of the Hawaii Conference on System Sciences*. January, 1986.

[3] Stuart I. Feldman.
Make -- A Program for Maintaining Computer Programs.
Software--Practice and Experience , April, 1979.

[4] P. G. Frankl and E. J. Weyuker.
A Data Flow Testing Tool.
In *Proceedings of the IEEE Softfair II*. San Francisco, December, 1985.

[5] Klaus Gewald.
Private Communication.
June, 1987.

[6] S. C. Johnson.
Lint, a C Program Checker.
In *Unix Programmer's Manual Supplementary Documents*. 4.2 Berkeley Software Distribution, 1984.

[7] Gail E. Kaiser and Peter H. Feiler.
An Architecture for Intelligent Assistance in Software Development.
In *Ninth International Conference on Software Engineering*, pages 80-88. IEEE, Monterey, CA, March, 1987.

[8] Butler W. Lampson and Eric E. Schmidt.
Organizing Software in a Distributed Environment.
In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1-13. June, 1983.

[9] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122-127. June, 1985.

[10] Harris M. Morgenstern.
An Inconsistency Management System.
Master's thesis, Columbia University Computer Science Department, March, 1987.

[11] John R. Nestor.
Toward a Persistent Object Base.
In Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), *Advanced Programming Environments*, pages 372-394. Springer-Verlag, Berlin, 1986.

[12] Leon J. Osterweil.
Toolpack -- An Experimental Software Development Environment Research Project.
IEEE Transactions on Software Engineering , November, 1983.

[13] S. Rapps and E. J. Weyuker.
Selecting Software Test Data Using Data Flow Information.
IEEE Transactions on Software Engineering 11(4):367-375, April, 1985.

[14] Barbara G. Ryder and Martin D. Carroll.
An Incremental Analysis Algorithm for Software Systems.
Technical Report CAIP-TR-035, Department of Computer Science, Rutgers University, March, 1987.

[15] Robert W. Schwanke and Gail E. Kaiser.
Smarter Recompilation.
ACM Transactions on Programming Languages and Systems , submitted for publication.

[16] *Debugging Tools for the Sun Workstation*
Sun Microsystems, Inc., 1986.

[17] Walter F. Tichy.
SmartRecompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[18] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden.
Ada-based Support for Programming-in-the-Large.
IEEE Software , March, 1985.

Change Management for Very Large Software Systems

Yoelle S. Maarek
Technion, Israel Institute of Technology
Computer Science Department
Haifa 32000, Israel

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Abstract

Very large software systems tend to be long-lived and continuously evolving. Purely managerial means for handling change are often adequate for small systems, but must be augmented by technological mechanisms for very large systems simply because no one person can understand all the interactions among modules. Many software development environments solve part of the problem, but most consider change only as an external process that produces new versions. In contrast, INFUSE concentrates on the actual change process and provides facilities for propagating changes that affect other modules. INFUSE structures the set of modules involved in a change into a *hierarchy of experimental databases*, where each experimental database isolates a collection of modules from the changes made to other modules and the hierarchy controls the integration of changes made to separate subsystems. The focus of this paper is on the clustering algorithm that automatically generates and maintains this hierarchy according to the strengths of interdependencies among modules as they are added and modified during development and maintenance.

To appear in **Seventh Annual International Phoenix Conference on Computers and Communications**, Scottsdale, AZ, March 1988.

1. Introduction

A Very Large Software System (VLSS) is composed of a large number of interdependent modules that typically undergo numerous changes during their lifetime. By *module*, we mean a separately compilable syntactic unit, such as an Ada^{TM} package, a Modula-2 module or a C source file. As such modules change, they often diverge from their specifications and the number of interface errors grows [12]. Change management tools are needed to coordinate programmers as they modify their modules, to propagate interface changes to dependent modules, and to enforce cooperation among programmers towards their goal of preventing interface errors. We describe a new algorithm that provides the basis for the INFUSE change management facility.

The change process in VLSS is considerably more complex than for small systems. For instance, determining the *extent* of a change (what is affected by the change) and its *implications* (what is necessary for restoring consistency after the change) is complicated by the sheer number of the interdependencies among pieces of the system. Moreover, an apparently simple change can easily *cascade* in unpredictable ways, requiring several rounds of changes for restoring consistency. Other problems such as the handling of temporary inconsistencies or the support of the iterative process of propagating changes become much more complex as the size of the system increases. INFUSE handles all these problems for *syntactic consistency*, that is, those inconsistencies that can be detected by a standard compiler; we are investigating extending INFUSE to semantic inconsistencies [14].

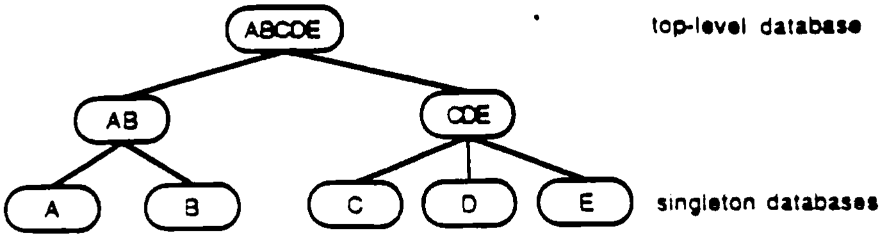
Several other tools have addressed simple cases of these problems. Make [3] automates recompilation of all dependent modules after source changes; it determines the extent of changes, and restores consistency by recompiling everything which might be affected, thus the first and fifth problems are solved in a rough way. Cedar's *System Modeller* [9] and Apollo's *Domain Software Engineering Environment* [10] (DSEE^{TM}) give programmers more control over dependencies among distinct versions of modules, but provide little more help than Make with respect to coordination and cooperation. None of these tools directly monitor the change process; DSEE permits each programmer to set up his own monitors to carry out specified actions whenever certain events occur, such as adding a new version to the baseline system. In contrast, INFUSE does not wait for deposit into the baseline system to perform its actions.

The NuMIL prototype [11] and Smile [6] are both much closer to INFUSE. The NuMIL prototype determines the impact of alterations based upon upward compatibility but provides analysis rather than control of the change process. Smile introduced the notion of an *experimental database*, which is a (virtual) copy of the baseline system that permits changes only to the subset of the system reserved by the user, isolating these changes from other programmers. INFUSE extends the notion of experimental database to a multiple-level hierarchy, and, unlike Smile, gathers *automatically* the modules into databases.

Previous papers on INFUSE have outlined its basic philosophy and discussed its automatic application of consistency-checking tools [15, 7]. In this paper, we briefly explain the INFUSE methodology and describe its use of a hierarchy of experimental databases for controlling and coordinating changes. We then present the algorithm INFUSE uses to automatically build and maintain this hierarchy.

2. The Hierarchy of Experimental Databases

INFUSE places all the modules involved in the change process in a distinguished experimental database: the *top level database*. This change set is normally chosen manually by a system analyst to attempt to satisfy the particular group of modification requests (MRs) appropriate for the next patch or release. Since the more numerous the modules in the change set, the more difficult the determination of the implications and the extent of changes, the top level experimental database is divided into several subsets that are themselves experimental databases. The implications and extent of changes in these smaller databases are easier to determine than in the top level one. By iteratively dividing the experimental databases into smaller and smaller databases, INFUSE limits the interactions that the programmers must cope with at one time. The hierarchy of experimental databases is the result of this division. The root of the hierarchy is the top level database, and each hierarchy level, from coarse to fine, is a partition of the original experimental database; a leaf contains a single module (see figure 1).



1. A hierarchy of experimental databases

The actual changes are made by editing the modules within their singleton databases. Once a singleton database is self-consistent it can be deposited into its parent database. An analysis tool is applied to determine this self-consistency: everything both defined and used within the module is used correctly with respect to its definition and everything used but not defined within the module is always used in a compatible manner. Once a singleton database is deposited, INFUSE coordinates and manages the iteration of changes by applying the following process recursively on every experimental database from the singletons to the top level (not included):

- When all child databases have been deposited into their parent, INFUSE invokes an analysis tool for performing change propagations within this parent database and checking the consistency among its subset of the changed modules. An analysis tool such as Lint [5] can be applied to the modules after all changes are made, or errors can be detected incrementally as by Mercury [8].
- If the database is self-consistent, then it can be deposited into its own parent database.
- If not, the local inconsistencies are detected and reported to the responsible programmers, who then negotiate and agree on new modifications for resolving the conflicts. The database, or only the part of it requiring further changes, is repartitioned into a subtree, and the singleton databases of that subtree are modified. The process above is reapplied to these experimental databases until the problematic database becomes self-consistent and can be deposited into its parent database.

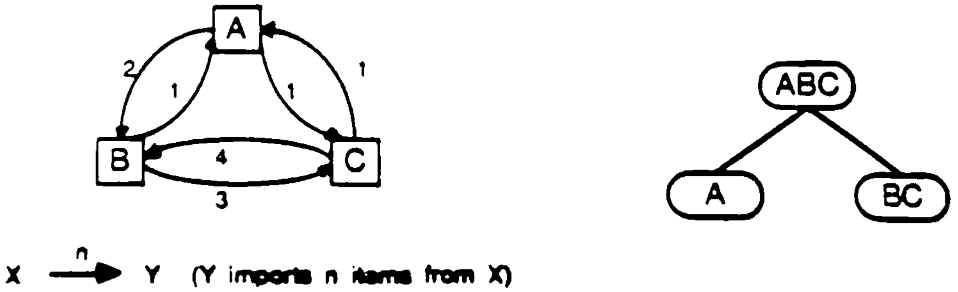
Finally, when all descendants have been deposited into the top level and it is both self-consistent and consistent with the modules of the baseline system that do not appear in the top level, the top level is itself merged back into the baseline.

The goal of this process is to support a widely accepted rule-of-thumb of software engineering: errors discovered early are much less costly to repair than those discovered late. The purpose of the hierarchy is to cluster together at the low levels those collections of modules where changes are most likely to lead to interface errors, ensuring early detection, and those collections of modules where the changes are unlikely to affect each other are not brought together until the high levels of the hierarchy.

Thus we need a measure for gathering collections of modules where changes are more or less likely to lead to interface errors. Our measure is the *interconnection strength* among pairs of modules, an approximation to the oracle that would tell us exactly how the future changes will effect other modules. Our approximation is based on the intuition that the probability of an interface error between modules M and

N is proportional to k, where module M uses i facilities imported from N, N uses j facilities from M, and k is the sum of i and j.

Consider three modules, A, B and C, importing and exporting items between each other, where an item is an importable syntactic unit of the programming language such as a procedure, a data type, *etc.* Since B and C are more strongly connected to each other than to A (see figure 2), they should be gathered in the same experimental database, A being added to them only at an upper level of the hierarchy.



2. Clustering according to the interconnection strength

3. Building a Hierarchy of Experimental Databases

There are two ways to build a hierarchy: top-down or bottom-up. The first way corresponds to *partitioning* methods and the second to *clustering*. In the partitioning approach we recursively divide the top level experimental database until reaching the singleton databases. When dividing a database, we need to know *a priori* the number of subsets we want to obtain: this approach is *model-driven*. Since the modules are available before beginning the construction of the hierarchy, we prefer the *data-driven* approach of clustering methods.

There is a strong analogy between the construction of a hierarchy of experimental databases and the hierarchical clustering of a set of objects. Clusters are groups of objects whose members are "more similar" to each other than to members of another group. The similarity between two clusters is measured by a *dissimilarity index*: the more similar any two clusters, the lower their dissimilarity index. There exist numerous hierarchical clustering algorithms [17] that differ only by the choice of the measure of similarity between clusters. Experimental databases correspond to clusters of modules, where the measure of similarity between clusters is the interconnection strengths between modules.

Hierarchical clustering is usually divided into two tasks: The first consists of applying the following general method [1] on the objects to be clustered.

- Identify the two clusters (initially a single object) that are the most similar according to the dissimilarity index.
- Merge them together into a single cluster.
- Repeat this process iteratively until there is only one cluster.

Every iteration in the clustering process forms a new *level clustering* by adding a new cluster and removing the merged clusters. The final output of the clustering process is often pictured as a hierarchy whose levels are these successive level clusterings; the hierarchy arises because each new cluster merges its two children in the immediately preceding level. The second task consists of selecting from this hierarchy the 'meaningful' level clusterings according to the needs of the application. This is usually done by an analyst since it requires knowledge of the application domain.

INFUSE expects a hierarchy where the arity of each experimental database is specific to the actual interconnection strengths of the modules in the change set. Our proposed algorithm combines the two tasks described above, without recourse to a human analyst; in particular, only the 'meaningful' level clusterings are actually generated, thus forming directly the hierarchy of experimental databases supported by INFUSE.

4. The Arity Controlled Clustering Algorithm

Unlike classical hierarchical clustering algorithms, our algorithm treats the level clusterings as temporary as long as they are not 'meaningful'. The temporary level clusterings are said to be *prospective*, whereas each level clustering that is selected is said to be *frozen*. The sequence of frozen level clusterings gives the hierarchy of experimental databases. To freeze level clusterings, the algorithm evaluates the similarity between the prospective level clusterings and an *exemplar*. We define the *arity* of an experimental database as its number of immediate descendants in the next level of the hierarchy. The similarity is computed by measuring the statistical dispersion of arities through a variance function defined as follows:

Let *LC* be a prospective level clustering and $\{x_1,x_2,\ldots,x_k\}$ the sequence of the arities of its k experimental databases; x_i represents the number of descendants that the i^{th} database of *LC* has in the previous frozen level clustering. The exemplar is defined by a single coefficient α. We define the measure v_α for evaluating the similarity between the *LC* and the exemplar by:

$$v_\alpha = \frac{1}{k}\sum_{i=1}^{k}(x_i-\alpha)^2$$

The initial frozen level clustering is composed of the singleton databases. Given this initial level clustering and an example arity for all the databases of the next level, the algorithm computes all the successive prospective level clusterings and freezes the one that minimizes our variance measure in order to determine the next level of the hierarchy. However, it is too costly to compute all the forthcoming level clusterings and to backtrack to the absolute optimum. In practice, the algorithm instead finds a local optimum, where the degree of locality is defined by a lookahead coefficient — that is, how many prospective level clusterings to generate.

The example arity is generated by the algorithm itself. It remembers past hierarchies involving the same software system, and uses previously successful values whenever possible. When not possible, such as in the early stages of the system's development when few changes have been made, the exemplar is chosen randomly or provided by an analyst.

Controlling the arity of experimental databases is reminiscent of the model-driven partitioning approach we rejected, where each partition splits an experimental database in a number of sets decided *a priori*. The similarity is misleading. When our algorithm controls the clustering arity of every level clustering, it treats this level arity as an exemplar that it is not necessary to meet. It chooses among several prospective level clusterings the one closest to the exemplar but does not force the construction of a level clustering identical to the exemplar.

We present a simplified version of our algorithm, with a lookahead equal to one, in figure 3. The overall time complexity of our algorithm is $O(n^2\log(n))$, the same as the classical clustering algorithms [16], even though we introduce supplementary computation by controlling the variance of the arities.

```
Input:    The interconnection strength values between pairs of modules.
          The coefficients $a,b,c,d$ for computing the interconnection strengths.
          The exemplar arity for every level clustering.
Output:   A hierarchy of experimental databases.

Start from the initial level clustering,
$L = \{\{m_1\},\{m_2\},\ldots,\{m_n\}\}$,
whose elements are the
singleton experimental databases reduced to a single module. Get the value of $\alpha$ for the next level. The current prospective level clustering is set to the previous frozen level clustering. The arity of each of its experimental databases is set to 1.

While there are more than two experimental databases in the current level clustering do:
    1. Construct the next prospective level clustering, NLC, by merging together the two experimental databases of the current level clustering that maximize $\delta$ (if there is more than one pair of clusters which realize this maximum, one of them is chosen arbitrarily. This new experimental database is their ancestor in the hierarchy.
    2. Update the interconnection strength values.
    3. If the $v_\alpha$ of NLC is greater than that of the current level clustering, freeze the current level clustering. The arities of the experimental databases of the current level clustering are set to one. Get the value of $\alpha$ for the next frozen level.
    4. Else the NLC becomes the current level clustering.
End While

Merge together the last two clusters of the current level clustering,
in order to form the last frozen level of the hierarchy.
```

3. The arity controlled clustering algorithm

The sequence of all the frozen level clusterings gives us the hierarchy of experimental databases.

5. Maintaining the consistency of the hierarchy

Changes made to modules may invalidate the hierarchy, in the sense that it no longer correctly reflects the interconnection strengths among modified modules. Two main classes of modifications can lead to invalidation:

1. Modifying the interface of a module, since the structure of the hierarchy is based on interconnection strength.
2. Adding a module to the hierarchy; a planned modification may involve creating a new module or conflict resolution may require modifying modules in the baseline but not in the original change set.

It is possible to treat a module whose interface has been modified in the same way as a new module. The older version is removed from the hierarchy, and the new one added. Therefore, we focus on adding a module to the hierarchy. The roughest way of updating the hierarchy is to recluster the entire change set, including the new module. This is too costly: Many experimental databases not affected by the modification would also be reprocessed, and deposits to these databases would have to be repeated. However, if we reject full reclustering and instead make only local changes, we cannot guarantee the resulting hierarchy is as 'good' as the one produced by our clustering algorithm. Fortunately, most practical cases (where relatively few interfaces are changed) affect only a small portion of the hierarchy and only this portion may not be the same as had full reclustering been applied.

In most cases, our *incremental* reclustering algorithm works as follows. The new module, M, is added to the top-level experimental database. Then it is merged into the next level experimental database with which it has the highest interconnection strength. This process is applied recursively until a singleton database is reached. The singleton is changed to contain two modules (the original and M) and has two new singleton children.

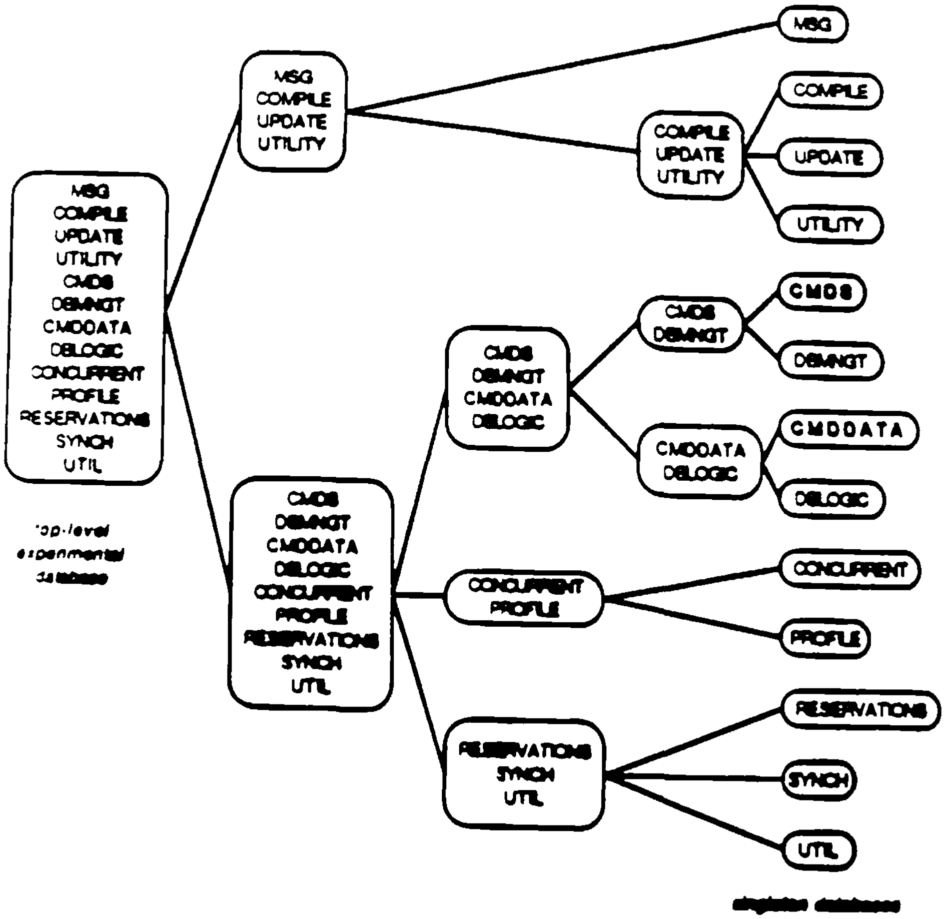
This naive algorithm works very nicely except for special cases where M is only weakly connected to each of the children of an experimental database, which occurs most frequently with a brand new module that is empty. Such a module is called an *outlier*. To determine that the module M is an outlier among several databases, $E_1,E_2, \ldots ,E_k$, our incremental algorithm computes the interconnection strength values between every pair of databases in the set: $\{E_1,E_2, \ldots ,E_k,\{M\}\}$. If the maximum is realized by a pair that does not include $\{M\}$, it means that M is less connected to any E_i than the E_i are interconnected among themselves. In this case, M is added as a new child of the parent experimental database.

6. Some empirical results

We selected Smile — a multiple-user programming environment for C developed as part of the Gandalf project [4] — as our test case for this paper since it is a medium sized system where the change processes involve few enough modules to be illustrated nicely in figures. We have also applied our clustering algorithm to the 60 modules of ALOE [2], also from the Gandalf project, as well as to several much smaller systems. Our example assumes that two Smile modules, CMDS and CMDDATA, are to be modified extensively. Therefore, the analyst also places the set of eleven modules related to them in the top-level experimental database, since these may also need to be modified. The interconnection strength values between these modules are automatically extracted from the program text and given in the following matrix (figure 4). Utility modules imported everywhere are not considered, since they are handled specially [15].

	CMDDATA	CMDS	COMPILE	CONCUR.	DBLOGIC	DBMGNT	RESERV.	UPDATE	UTILITY	MSG	PROFILE	SYNCH	UTIL
CMDDATA	x												
CMDS	27	x											
COMPILE	17	8	x										
CONCUR.	2	13	3	x									
DBLOGIC	31	23	31	4	x								
DBMGNT	42	51	10	1	25	x							
RESERV.	18	9	0	0	1	6	x						
UPDATE	5	21	52	3	30	46	2	x					
UTILITY	27	21	28	2	0	6	0	31	x				
MSG	1	0	0	0	0	0	0	0	0	x			
PROFILE	0	2	1	3	0	2	0	0	3	0	x		
SYNCH	2	5	0	0	0	4	1	0	0	0	0	x	
UTIL	18	17	9	0	5	19	4	4	14	3	1	6	x

4. Matrix of interconnection strengths



5. Hierarchy of experimental databases for Smile

Given this data, our algorithm produces a hierarchy (see figure 5) similar to the one manually identified by a Smile 'expert'. When applied to the larger ALOE, the hierarchies obtained are still very similar but not identical to the ones computed by hand.

7. Conclusion

We have described INFUSE, a software development environment that supports change management in addition to recompilation and version control after changes. Unlike other tools, INFUSE assists programmers during rather than after the change process. Conflicts are detected early when they are relatively inexpensive to repair, rather than later after the entire change process has completed and recompilation and testing has begun. The major contribution of this paper is the presentation of a new clustering algorithm which makes such conflict detection and resolution possible. From the change set, INFUSE automatically builds a hierarchy of experimental databases where the most strongly connected modules are collected together into the 'natural' clusters specific to the VLSS and negotiation of module interface errors are enforced. INFUSE thus provides practical support for managing and coordinating changes in very large software systems. We are currently extending INFUSE with mechanisms to combine stubs and test drivers hand-constructed for unit testing to operate as test harnesses for the integration among strongly connected clusters of modules.

Acknowledgments

Dewayne Perry and Gail Kaiser first developed INFUSE in the context of the Inscape project [13] at AT&T Bell Laboratories, where Peggy Quinn was responsible for extracting the Smile and ALOE dependency matrices. Travis Winfrey, Ben Fried and Pierre Nicoli completed the implementation under the direction of Bulent Yener. Galina Dastkovsky, Michael Elhadad, Steve Popovich and the anonymous referees read earlier versions of this paper and made useful criticisms and suggestions.

This research was supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer and Information Systems, and in part by a Digital Equipment Corporation Faculty Award.

References

[1] E. Diday, J. Lemaire, J. Pouget and F. Testu.
Elements d'Analyse des Donnees.
Dunod, 1982.

[2] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

[3] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[4] A.N. Habermann and D. Notkin.
Gandalf: Software Development Environments.
IEEE Transactions on Software Engineering SE-12(12):1117-1127, December, 1986.

[5] S.C. Johnson.
Lint, a C Program Checker.
Unix Programmer's Manual.
AT&T Bell Laboratories, 1978.

[6] Gail E. Kaiser and Peter H. Feiler.
Intelligent Assistance without Artificial Intelligence.
In *Thirty-Second IEEE Computer Society International Conference*, pages 236-241. San Francisco, CA, February, 1987.

[7] Gail E. Kaiser and Dewayne E. Perry.
Workspaces and Experimental Databases: Automated Support for Software Maintenance and Evolution.
In *Conference on Software Maintenance*, pages 108-114. Austin, TX, September, 1987.

[8] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[9] Bulter W. Lampson and Eric E. Schmidt.
Organizing Software in a Distributed Environment.
In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 1-13. San Francisco, CA, June, 1983.
Proceedings published as *SIGPLAN Notices*, 18(6), June 1983.

[10] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[11] K. Narayanaswamy.
A Framework to Support Software System Evolution.
PhD thesis, University of Southern California, May, 1985.

[12] D.E. Perry and W.M. Evangelist.
An Empirical Study of Software Interface Errors.
In *International Symposium on New Directions in Computing*, pages 32-38. Trondheim, Norway, August, 1985.

[13] Dewayne E. Perry.
Programmer Productivity in the Inscape Environment.
In *IEEE Global Telecommunications Conference*, pages 428-434. December, 1986.

[14] Dewayne E. Perry.
Software Interconnection Models.
In *9th International Conference on Software Engineering*, pages 61-69. Monterey, CA, March, 1987.

[15] Dewayne E. Perry and Gail E. Kaiser.
Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems.
In *ACM Fifteenth Annual Computer Science Conference*, pages 292-299. St. Louis, MO, February, 1987.

[16] J.C. Simon.
Etudes et Recherches en Informatique: La Reconnaissance des formes par algorithmes.
Masson, 1985.

[17] R. R Sokal and P. H. Sneath.
Principles of Numerical Taxonomy.
W. H. Freeman, 1973.