ers for asynchronous communication networks," in *Proc. 2nd Int. Workshop Distrib. Algorithms*, Amsterdam, The Netherlands, July 8-10, 1987.
[16] A. Segall, "Distributed network protocols," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 23-25, Jan. 1983.
[17] P. Spirakis and A. Tsakalidis, "A very fast, practical algorithm for finding a negative cycle in a digraph," in *Proc. ICALP 86* (Lecture Notes Comput. Sci., Vol. 226). Berlin, West Germany: Springer-Verlag, 1986, pp. 397-406.

An Efficient Distributed Knot Detection Algorithm

ISRAEL CIDON

***Abstract*—A distributed knot detection algorithm for general graphs is presented. The knot detection algorithm uses at most $O(n \log n + m)$ messages and $O(m + n \log n)$ bits of memory to detect all knots' nodes in the network (where n is the number of nodes and m is the number of links). This is compared to $O(n^2)$ messages needed in the previous published best algorithm. The knot detection algorithm makes use of efficient cycle detection and clustering techniques.**

Various applications for the knot detection algorithms are presented. In particular, we demonstrate its importance to deadlock detection in store and forward communication networks and in transaction systems.

***Index Terms*—Clustering, cycle detection, deadlock detection, distributed algorithms, knot detection.**

I. Introduction

A knot in a directed graph is a strongly connected subgraph with no edge directed away from the subgraph. A knot is a useful concept for describing deadlocks in computer systems. In [1], [2], deadlocks in store and forward networks are described as knots in the buffer graph of the network. The concept of a knot in the buffer graph is also used for deadlock resolution techniques when the deadlock is resolved by discarding packets at nodes. The minimum number of packets that should be discarded in order to resolve the deadlock is exactly one packet in each knot [3], [4]. Knots were also found to be useful for representing deadlocks in transaction systems [5], [6].

A distributed knot detection algorithm is the basis for developing a distributed deadlock detection algorithm. In [3], a deadlock detection algorithm for buffer deadlock in store and forward networks is described which is based on a knot detection for a general graph. The algorithm developed here can replace the knot detection of [3], resulting in a more efficient deadlock detection algorithm.

Various distributed knot detection algorithms have been suggested in the literature. Some of them are imbedded in more general deadlock detection algorithms. Three basic classes of algorithms have been suggested.

1) Collecting the complete graph topology at each node and detecting the knot by each individual node [2].

2) Testing individually at each node whether it is a member of a knot using a search algorithm. A distinct search is used for each node [4]-[6].

3) Using cycle detection and clustering technique in which cycles of clusters are detected and merged into bigger clusters [3].

Manuscript received May 27, 1987; revised May 17, 1988.
The author is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.
IEEE Log Number 8926734.

Class 1) is the most inefficient in terms of total number of messages and bits sent and the amount of memory needed to support its operation. Here, the graph topology is collected at each node by means of flooding, resulting in communication cost of $O(nm)$ messages and a total of $O(m^2)$ bits (where n is the number of nodes and m the number of edges). The memory required at each node is $O(m)$ bits, resulting in a total of $O(nm)$ bits. However, this algorithm is very simple and very fast.

A more efficient algorithm can be developed using 2). Here, since for each node a complete search in the graph is performed, $O(m)$ messages are needed to test if a single node belongs to a knot. For detecting all nodes, $O(nm)$ messages are needed. However, comparing to 1), each message is only $O(\log n)$ bits long (stamped with the origin node identity) and the total memory needed in the network for this algorithm is $O(n^2 \log n)$ bits. This technique is considerably more complex than that of 1).

Using the third technique, in [3], the total number of messages is reduced to $O(n^2)$ in the worst case, each of $O(\log n)$ bits and a total of $O(m + n \log n)$ bits of memory are needed. This improvement in the communication and the memory costs is accomplished by considerably increasing the complexity of the algorithm and reducing its speed. In [4], it is explained why a low communication cost and especially a low memory cost deadlock detection algorithm are invaluable in the environment of buffers deadlock in store and forward networks. In such a network, a deadlock situation occurs when too many packets are waiting to be served by the network while there is not enough memory to accomplish this service. Deadlocks occur under heavy load of traffic and shortage of memory. This is the main motivation for developing protocols which use fewer messages and less and memory at the expense of complexity and speed.

The algorithm of this paper belongs to the third class. We succeed in further improving the efficiency of the knot detection by employing phase numbers in the spirit of [7]. The total number of messages needed is reduced to $O(m + n \log n)$, each of $O(\log n)$ bits and the total number of memory bits is $O(m + n \log n)$.

In Section II, we give the model of the system and the definition of a knot. In Section III, we describe the outline of the new knot detection algorithm. In Section IV, a detailed description of the algorithm is given. In Section V, the communication and the memory costs are evaluated.

II. The Model

A *network* consists of a set of *communication nodes N* and a set of bidirectional *communication links L* that interconnect the nodes of *N*.

Regarding links, the following properties are assumed. They are FIFO (do not lose, reorder, or duplicate messages); there is no bound on the amount of time that it takes a message to traverse a link; any message placed on the link arrives at the other side of the link in finite time; links never fail.

We assume that at each node i, each attached link l may be designated as an *outgoing link*. (In deadlock detection, this implies that there is a request pending for this specific link.)

Let (V, E) be a directed graph where $V = N$ is the set of vertices in the graph and E a set of directed edges where a directed edge (i, j) indicates that in node i the link (i, j) is designated as an outgoing link. A *tie* T in (V, E) is a set of nodes with no links directed from T to $N - T$.

A *knot* K is a tie of which any subset is not a tie. This implies that K is a set of strongly connected nodes. Alternatively, node i is a member of a knot if i is reachable from all nodes which are reachable from i. In that case, the knot is the set of nodes which are reachable from i (including i itself). Obviously, any tie contains at least one knot. In Fig. 1, an example for a knot and ties is depicted.

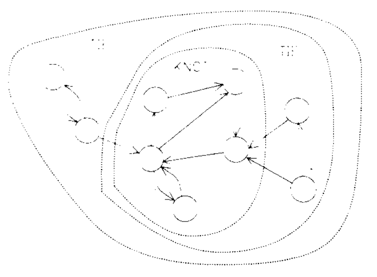


Fig. 1. Knot and ties.

III. Overview of the Algorithm

In this section, we describe the basic ideas of the knot detection algorithm. The goal of the algorithm is to finally assign to each node a value called the state of the node. The final state can be either *KNOT*, indicating that this node is a member of a knot, or *FREE*, indicating it is not a member of any knot.

The detection of a knot consisting of a single node is trivial since such a node does not have any outgoing link. Consequently, we focus on the detection of knots with more than one node. We assume that all single node knots are immediately detected and the state of these nodes is set to *KNOT*.

The algorithm exploits the property that nodes of a knot are strongly connected, and therefore a knot (with more than one node) contains a *cycle* of nodes. A group of nodes which are found to be strongly connected is called a *cluster*. The algorithm is based on looking for cycles of clusters and merging them into bigger clusters. Since each of the clusters is strongly connected and a cycle is a strongly connected subgraph, the new cluster, formed by merging all the clusters of a cycle, is strongly connected as well. If a cluster with no link directed outside the cluster is detected, then a knot is found. At the beginning, each cluster consists of a single node. At the end of the algorithm, each cluster contains all nodes of a knot.

The algorithm consists of two basic steps which are repeated to the end. First, within each cluster, a single outgoing (outgoing from the cluster) link is selected. In the second step, it is checked whether this link is directed to either a *KNOT* or a *FREE* state node. If affirmative, then the states of all nodes of the cluster are set to *FREE*. These nodes will not be involved in any further action. We repeat this procedure until all possible nodes are set to *FREE*. In the case that some clusters have outgoing selected links which are not directed to a *KNOT* or *FREE* node, then a cycle of clusters and selected outgoing links exists and detected. All clusters of this cycle are merged into a single cluster. These two steps are repeated until all nodes are either *FREE* or some clusters without any outgoing links are found. In the latter case, the states of all nodes of these clusters are set to *KNOT*. In Fig. 2, an example to this procedure is given. The solid arrows represent selected outgoing links, the dashed arrows represent outgoing links, and the dashed circles represent clusters. In (a), node 1 is detected to be a knot while all the rest select a single outgoing link. In (b), nodes 2, 3, and 4 are merged to a single cluster. The same is true for nodes 5 and 8. The cluster of nodes 2, 3, and 4 is directed to a knot and thus no longer considered. In (c), 6 is added to the cluster of 5 and 8. In (d), a knot containing 5, 6, 7, and 8 is detected. Consequently, nodes 9 and 10 transit to the *FREE* state.

The important parts of the algorithm are the detection of cycles of clusters and the technique to combine a cycle of clusters to a single cluster. The cycle detection is facilitated by having nodes/clusters pass composed ID's (to be later defined) through the links of the cycle and have the maximum ID node receive back its own ID. The cycle combination is facilitated by having this maximum ID node establish a tree structure whereby all nodes of the cluster report to it.

We start by describing the basic ideas of the cycle detection part. We assume that each cluster involved in this part consists of a single node or "acts" as a single node. We will later describe how this is accomplished in a distributed manner.

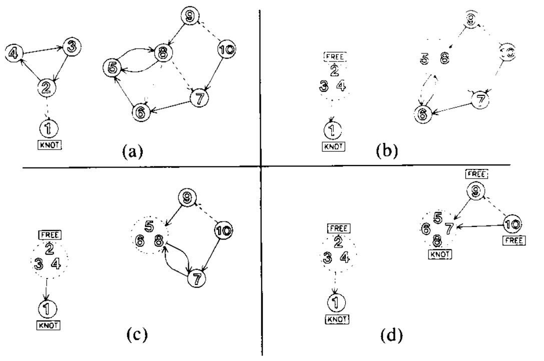


Fig. 2. Cluster merging procedure.

The cycle detection is started at a node i by choosing an arbitrary outgoing link l and designating it as a selected outgoing link. The node to which this link is incoming is called the *father* of i and i is called a *son* of this node. Since a single outgoing link is selected, each node may have several sons, but only one father. If the father of node i is in the *KNOT* or the *FREE* state, node i sets its state to *FREE*. This procedure eventually causes each node which has a directed path of selected outgoing links to a *KNOT* node to become *FREE*. In Fig. 3, we demonstrate this procedure.

After the assignment of the *FREE* states, the remaining group of nodes and selected outgoing links must include at least one cycle. Consider one of these cycles along with all nodes that have a path of selected outgoing links to this cycle. In Fig. 4, an example to such a subgraph is depicted.

In order to detect the cycle, a simple algorithm is suggested in [2]. The nodes forward maximal identities to their sons. Whenever a node receives an identity which is higher than any other identities received by this node (including its own identity), it is recorded and sent to all sons. Whenever a node receives its own identity from its father, it realizes that it belongs to a cycle and has the maximal identity in that cycle. This information can now be forwarded to all other members of the detected cycle through the selected outgoing links. In Fig. 5, a simple detection procedure is demonstrated for the cycle of Fig. 4.

The major drawback of this simple algorithm is that the number of ID's sent by each node may be of the order of the number of nodes in the subgraph, which results in a quadratic order for the total communication cost. In order to reduce the worst case communication cost, node identities should be more carefully forwarded by the nodes. As in [7], we try to guarantee that whenever an identity is forwarded, it will "cover" enough nodes so that the total number of ID's forwarded by a node will be in the worst case a logarithmic order of the number of nodes. For this purpose, we introduce the notion of a *composite identity* (*CID*). A composite identity consists of two numbers, a phase number *PN* (which is later described) and a node cluster identity *NCD*. Composite identities are compared lexicographically. We say that CID_1 is *strictly smaller* (larger) than CID_2 if PN_1 is smaller (larger) than PN_2. It is *simply smaller* (larger) if $PN_1 = PN_2$ and NCD_1 is smaller (larger) than NCD_2. Whenever the father's composite identity is found to be strictly larger than that of the son, the composite identity is accepted by this son.

Generally, this procedure is not enough for covering all nodes of the cycle with the same composite identity. It terminates whenever all nodes have the same phase number (but not necessarily the same node identity). In the following, we describe how phase numbers are incremented in order to cover the entire cycle with the same composite identity. Let us denote by a *segment* the group of all nodes (clusters) that are covered by the *same composite identity*.

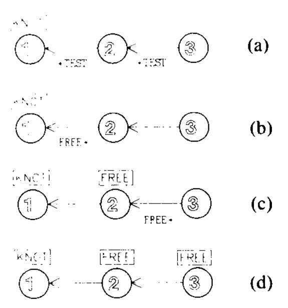


Fig. 3. *FREE* messages propagation.

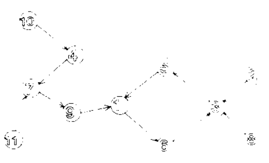


Fig. 4. Network of selected outgoing links.

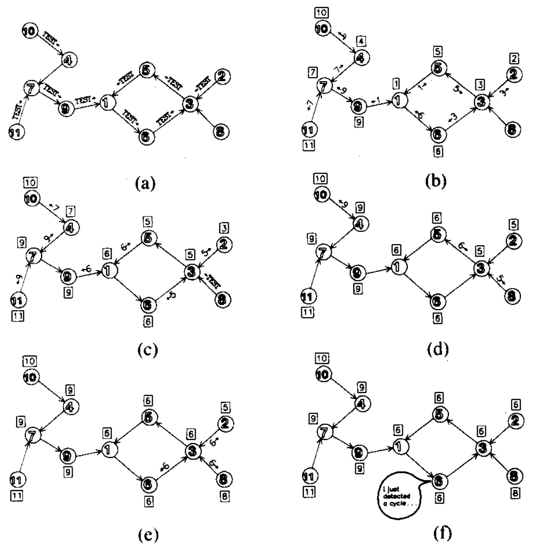


Fig. 5. A simple algorithm for cycle detection.

As is later shown, a segment is structured as a directed tree of selected outgoing links. The root of the segment is the node (cluster) whose identity covers the nodes of the segment. The father of a segment is the father of its root node; the sons of the segment are the sons of its leaf nodes. A segment is a *local maxima* segment if its father and at least one of the sons of the segment have composite identities simply smaller than its composite identity. Whenever a local maxima is detected, its phase number is incremented by one by its root node. The new composite identity replaces the current one at all the nodes of the segment and the previous procedure is repeated. By letting only local maxima segments increment their phase number we guarantee: 1) the new composite identity will cover at least one additional segment currently covered by the same (previous) phase number, 2) this local maxima segment will not be covered by a different composite identity with the new phase number (since the father's node identity is smaller than this segment's node identity).

Consequently, assume that a segment covered by a phase number n consists of at least 2^n nodes (which is clearly true for $n =$

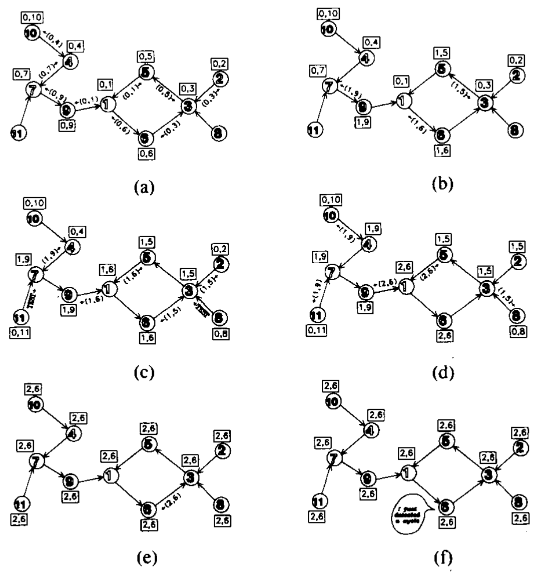


Fig. 6. Cycle detection using segmentation.

0); it is easy to prove by induction using 1) and 2) that a segment covered by the phase number $n + 1$ contains at least $2 \times 2^n = 2^{n+1}$ nodes. Since for each phase number only a single composite identity is accepted, this bounds the number of different composite identities accepted by a node to a logarithmic order of the total number of nodes. Cycle detection is a byproduct of this algorithm. A cycle is detected whenever the root node detects that its father is covered by its own composite identity. A cycle detection using this procedure is demonstrated in Fig. 6.

Another important part of the algorithm is the distributed procedure to achieve coordination between nodes of clusters and segments. The coordination of a segment is straightforward since each segment forms a rooted tree. The decision whether to increment the phase number is taken at the root node (or cluster) of the segment. Since composite identities are forwarded from fathers to sons, the composite identity of the segment's father is accepted by its root node. The root node informs its father when its identity is found to be simply smaller than that of the father. The leaf nodes report the root node upon positive comparison results at their sons (which are root nodes of other segments) through the rooted tree. If the root node decides to increment the phase number, it broadcasts the new composite identity to all the nodes of the segment upstream the rooted tree.

The coordination and merging of clusters are similar but more complex than the coordination of segments. Basically, in each cluster, a directed control tree is maintained. This tree is rooted at a special node called the *cluster leader*. The cluster leader serves as a central control point of the cluster to which the needed information is brought and in which decisions are made. The cluster leader also coordinates the search for a new outgoing link in the cluster. Not all the processes in the cluster are controlled by the cluster leader. In order to save communication, some local decisions are taken by nodes of the cluster without informing the leader.

IV. Detailed Description of the Algorithm

We start by describing the algorithm for clusters that contain a single node, and then generalize it to arbitrary size clusters.

The cycle detection is started at node i by choosing an arbitrary outgoing link l, designating it as a selected outgoing link (SO_i), and sending a *TEST* message over it. If no such outgoing link exists, the node sets its state to *KNOT*. If l is directed to a *KNOT* or a *FREE* node, this node replies with a *FREE* message. Whenever a *FREE* message is received by a node which is not *FREE*, the node sets its state to *FREE*, and forwards the *FREE* message over its selected incoming links (to be described). This flooding of *FREE* messages causes every node which has a directed path to a *KNOT* node to become *FREE*.

Any non-*FREE* or *KNOT* node receiving a *TEST* message over a link *l* designates this link as a *selected incoming link* (member of *SI*) and sends back over this link its current composite identity (*CID*). *CID* is a composite identity vector containing the following information:

1) the node phase number (*PN*) which is the maximal phase number heard by this node (initially 0),

2) the node cluster number (*NCD*) which identifies the leader of the cluster (initially the node's own identity),

3) membership flag (*MF*) which indicates if the node has already joined the cluster whose leader is *NCD* (initially 1).

We say that CID_1 is strictly smaller than CID_2 if $PN_1 < PN_2$ and simply smaller if $PN_1 = PN_2$ and $NCD_1 < NCD_2$. Similarly we define the terms strictly larger and simply larger.

Whenever a new *CID* is received at node *i*, it is compared to the node's current *CID*. If the received one is strictly larger than the current one, it is accepted and replaces the current *CID*. In addition, the leadership flag LF_i and the membership flag MF_i are set to zero. (This indicates that the node is not the root node of the segment covered by the new *CID*.) The new *CID* is sent over all selected incoming links.

In order to increment its phase number, a node (and later on a cluster leader) must have both its father and one of its sons *CID*'s to be simply smaller than its current *CID*. The variables *MAX*2 and *MAX*1 are used to record these events, respectively.

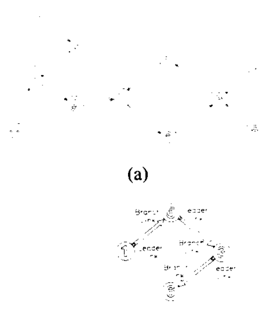
If the received *CID* is simply larger than the current one, the node sends an *INC*1(*PN*) message back to the sender. The node also sets the variable *MAX*2 to 0, indicating that its segment is not a local maxima. If the received *CID* is simply smaller than the current one, then the node sets the variable *MAX*2 to 1. If it is strictly smaller, this message is ignored.

Upon receiving an *INC*1(*PN*) message at a node having its leadership flag set to 1 and the received *PN* is equal to its current phase number, the node sets the variable *MAX*1 to 1. If the same conditions hold with the exception that the leadership flag is zero, then the node passes this *INC*1(*PN*) over its selected outgoing link. Whenever the variables *MAX*1 and *MAX*2 are both equal to 1, the node increments its phase number by one and broadcasts its new *CID* over all selected incoming links. Both *MAX*1 and *MAX*2 are then reset to zero.

Since *CID*'s are sent over incoming links, the existence of a cycle implies that one node (the node with the maximal *CID* in that cycle) will receive its own *CID* back over its selected outgoing link. When this happens, this node detects a cycle and appoints itself to be a cluster leader.

Next we describe how the leader combines the detected cycle into a cluster. In this process, the leader builds a control tree rooted at itself. This tree is used to coordinate the cluster activities, i.e., to make it "act" like a single node. Through this tree, messages are exchanged between the leader and the other members of the cluster.

First, the leader sends a message (*JOIN*(*i*)) stamped with its own identity over its *SO* link. A node receiving this message over an *SI* link records its leader's identity, deletes this link from the list of selected incoming links, and designates it as a *leader link* (*LL*). Messages to the leader are forwarded over this link. The membership flag is set to 1. The node forwards the *JOIN* message over its selected outgoing link, and designates this link as a *branch link* (member of *LB*). Messages from the leader are forwarded over the branch links. Finally, the *JOIN* message arrives through the cycle back to the leader. The link on which the leader receives this message is not a part of the leader tree. The leader deletes this link from the list of selected incoming links and sends a *DELETE* message over this link. Upon receiving this *DELETE*, the receiver


(a) (b)
Fig. 7. A typical snapshot.

deletes this link from *LB*, and sends a completion message (*JOIN_TERM*) over its leader link. Any node receiving this *JOIN_TERM* message over all *LB* links (here, *LB* consists of a single link) forwards it over *LL*. In Fig. 7, we demonstrate a typical snapshot of the algorithm. In (a), the nodes with their selected outgoing links are depicted. In (b), the resultant cluster along with its leader tree is depicted.

The basic difference between trees which are the result of merging a cycle of single nodes and trees of general clusters is that in the first case, each node has only one branch link and in the second case, each node may have many branch links. In both cases, the leader tree connects all nodes of the cluster. In each node, the identities of the links that belong to the tree are recorded. The node distinguishes between the leader link (*LL*) which leads to the leader (the root node of the tree) and the branch links (*LB*) which leads to the leaves of the tree. In Fig. 8, an example of such a structure is depicted.

Next, we describe how clusters are coordinated to detect a cycle of clusters. After a new cluster is formed, a single link, outgoing from the cluster, is chosen. The search for such a link is done using a distributed depth-first search through the leader tree, starting at the leader. A node presently active in the search process (called the *active node*) looks for an untested outgoing link (i.e., a *TEST* message was never sent over it). If all links are tested, the active node passes the search deeper into the tree by sending a search message (*SEARCH*) over one of its branch links from which a search termination message (*SEARCH_TERM*) has not been received. This passes the activity to another node.

If an untested link is found, a *TEST* message is sent over it (the link becomes the cluster outgoing link). If the identity of the present leader is received over this outgoing link along with a flag indicating it is the leader of the sender, then this link connects two nodes of the same cluster. A *DELETE* message is sent over it in order to inform the node at the other end to delete it from *SI*. This *DELETE* message is acknowledged by a second *DELETE* message. Upon receiving this acknowledgment, the node designates the link as tested, and proceeds with the search.

When a node has received *SEARCH_TERM* messages over all its branch links (or it does not have branch links or untested links), it sends a *SEARCH_TERM* message back over its *LL* to "back up" the search. If the leader receives such messages over all its branch links, it determines that its cluster forms a knot. The leader then informs all nodes in its cluster of this situation by broadcasting a *KNOT*. Upon receiving such a message over its leader link, a node forwards the *KNOT* message over its branch links, sets its state to *KNOT*, and sends a *FREE* message over all its *SI* links. Note that all the links of the cluster were tested and all links that were found to connect nodes of the same cluster were deleted from *SI* list (either after receiving a *DELETE* message during the search or when the link was designated as a leader link or a branch link). This implies that all the remaining *SI* links are incoming from nodes external to this cluster and thus are not members of the knot.

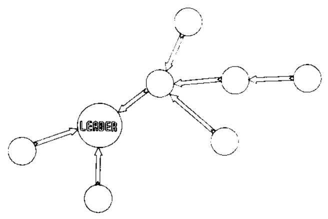


Fig. 8. A typical cluster.

The other possible events after an untested link is chosen to be the cluster outgoing link are as follows.

A new *CID* is received over this link. The active node compares it to its current *CID* and acts in the following manner depending on the cases below.

1) If the received *CID* is strictly smaller than its current *CID*, then the node ignores this message.

2) If it is simply smaller, the node sends an *INC*2(*PN*) message to the leader containing the current phase number. This message is forwarded through the leader tree to the leader. Upon receiving the *INC*2(*PN*) message with the current phase number, the leader sets the variable *MAX*2 to 1. If both *MAX*1 and *MAX*2 are equal to 1, the leader increments its phase number by one and broadcast the new *CID* over all branch and selected incoming links. All the nodes of the cluster follows the same acation.

3) The received *CID* is equal to the node's current *CID* with the exception of a zero leadership flag. Here, a cycle of clusters has been detected. At this point, all clusters of that cycle are combined into a single cluster whose leader has the identity just received. The process is to combine the leader trees of the clusters in the cycle into a single tree, rooted at the new leader. The node which is responsible for performing this cluster combination is the active node (the node adjacent to the cluster selected outgoing link). The new part of the leader tree will be rooted at this node. The active node starts the construction by sending a *JOIN* message stamped with the leader identity. A node which receives such a *JOIN* message over a link changes its leader identity, sets its leadership flag to 1, designates this link as the new leader link, designates all other old leader tree links and the cluster selected outgoing link (if any) as branch links, and forwards the *JOIN* message over all the branch links. This changing leader process is terminated by a *JOIN_TERM* message flowing back through the new *LL*'s to the node that initiated the process. After the active node has received the *JOIN_TERM* message over the cluster outgoing link, it designates this link both as a branch link and as a tested link, and continues the search for a new cluster outgoing link (the search is forwarded to the new branch link as well). In Fig. 9, we demonstrate this cluster combining process. In (a), the clusters are depicted along with their leader trees and selected outgoing links. In (b), the resulted combined cluster is depicted.

4) If the received *CID* is simply larger than the current one, the active node responds with an *INC*1(*PN*) message. This tells the sender that its *CID* potentially forms a local maxima.

A more complex situation occurs when the active node receives a *CID* which is strictly larger than its current one. In this case, the cluster should be covered by this new *CID* and should join the segment whose root node (cluster) has the node identity just received. This new *CID* must be broadcast over all selected incoming links of this cluster, and compared to the neighboring nodes' *CID*'s. The comparison results should be forwarded back over the cluster selected outgoing link. To facilitate this procedure, we replace the root node of the control tree (currently the leader of the cluster) by the active node. It is not important any more to keep the regular structure of the cluster since it is already known that this cluster is not the root of the segment. This cluster will not be expanded any more. It may only join other clusters or become *FREE*. Consequently, it is more convenient to root the tree at the active node

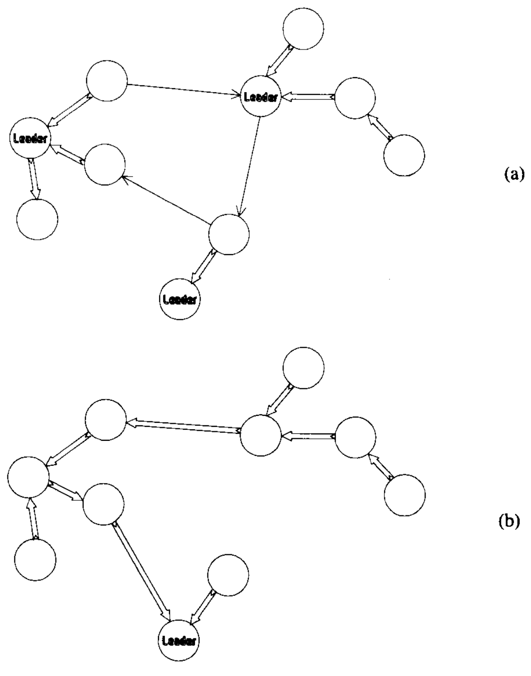


Fig. 9. Cluster merging.

which is attached to the cluster selected outgoing link. The active node replaces its *CID* with the received one, resets its membership flag to zero, and broadcasts its new *CID* over all branch links and over all *SI* links.

Upon receiving a new *CID* strictly larger then its current one over a particular link, a node (which is not active) designates that link as a leader link, designates the old leader link, if it is different from the current one, as a branch link, resets its membership flag, replaces its current *CID*, and broadcasts it, as described for the active node. In this case, the search for a new outgoing link in the cluster is stopped.

When a node receives a *FREE* message over a link, it follows the same procedure except that now a *FREE* message replaces the previous *CID* message. The node also sets its state to *FREE*. The algorithm ends when all nodes are in one of the two states *FREE* or *KNOT*.

In [10], a formal description of the algorithm is presented in the form of pseudocode. It is omitted here in order to save space.

V. Cost

Communication Cost

To compute the communication cost, measured in number of messages sent in the network, we consider separately each message type. Let n be the number of nodes in the system, and let m be the number of unidirectional links which are attached to those nodes.

1) *TEST* and *FREE* messages can be sent once over any unidirectional link, and thus at most $2m$ such messages are sent.

2) *INC*1, *INC*2, *JOIN*, *JOIN_TERM*, *SEARCH*, and *SEARCH_TERM* messages are sent only over trees or cycles (and each cycle with the exception of one link eventually becomes part of the final tree), once for each phase number. Since there are no more than $\log n$ phases, the total number of transmissions is bounded by $12n \log n$.

3) *MAXCID* messages are received over selected outgoing links. However, for a node which is not in a knot, such a link is only selected once (a single link per node). For nodes in knots, such messages can be received only over outgoing links which are part of a cycle (except if the link is found to connect two nodes of the same cluster, which may occur only once and then it is the only

message sent over that link), and thus no more then $n \log n + m$ of such messages can be sent.

We can conclude that communication cost at worst case is $O(m + n \log n)$ messages, each of no more than $O(\log n)$ bits.

Memory Cost

In order to compute the memory cost of the algorithm, we assume that one bit variable is allocated for describing the membership of each adjacent link to each set of links (O_i, SI_i, T_i, SO_i, LL_i, LB_i). This implies that except for CID_i, which contains a node identity, a phase number, and a binary flag, all variables are of one bit length, and a fixed number of such variables are allocated for each link adjacent to a node.

We can conclude that the total memory cost of this algorithm is $O(n \log n + m)$ bits for the total network, and $O(\log n + k)$ for each node where k is the degree of that particular node.

Acknowledgment

The author would like to thank J. M. Jaffe, M. Sidi, and I. S. Gopal for helpful discussions and the anonymous reviewers for their helpful comments and suggestions.

References

[1] K. D. Gunther, "Prevention of deadlocks in packet-switched data transport systems," *IEEE Trans. Commun.*, Special Issue on Congestion Control in Computer Networks, vol. COM-29, pp. 512–524, June 1981.

[2] G. Gambosi, D. P. Bovet, and D. A. Menascoe, "A detection and removal of deadlocks in store and forward communication networks," in *Performance of Computer-Communication Systems*, H. Rudin and W. Bux, Ed. Amsterdam: Elsevier Science B.V. (North-Holland), 1984, pp. 219–229.

[3] I. Cidon, J. M. Jaffe, and M. Sidi, "Local distributed deadlock detection by cycle detection and clustering," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 3–14, Jan. 1987.

[4] ——, "Local distributed deadlock detection with finite buffers," in *Proc. IEEE INFOCOM'86*, Miami, FL, Apr. 1986, pp. 478–486.

[5] J. Misra and K. M. Chandy, "A distributed graph algorithm: Knot detection," *ACM Trans. Programming Lang. Sys.*, vol. 4, pp. 678–686, Oct. 1982.

[6] G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection," in *Proc. Symp. Principles of Distributed Comput.*, Oct. 1984, pp. 285–301.

[7] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum weight spanning trees," *ACM Trans. Programming Lang. Syst.*, vol. 5, pp. 66–67, Jan. 1983.

[8] P. A. Humblet, "A distributed algorithm for minimum weight directed spanning trees," *IEEE Trans. Commun.*, vol. COM-31, pp. 756–762, June 1983.

[9] E. Gafni and Y. Afek, "Election and traversal in unidirectional networks," Dep. Comput. Sci., UCLA.

[10] I. Cidon, "An efficient distributed knot detection algorithm," IBM Res. Rep. RC 12099, Aug. 1986.

An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort

RANDY K. LIND AND K. VAIRAVAN

Abstract—**In this paper, we report the results of an experimental study of software matrices for a fairly large software system used in a real-time application. We examine a number of issues, including the mutual relationship between various software metrics and, more importantly, the relationship between metrics and the development effort. We report some interesting connections between metrics and the software development effort.**

Index Terms—**Development effort, metrics, real-time software, software complexity.**

Manuscript received May 29, 1987; revised December 22, 1987.

R. K. Lind was with General Electric Medical System, WI. He is now with NMT Corporation, Lacrosse, WI.

K. Vairavan is with the Department of Electrical Engineering and Computer Science, University of Wisconsin, Milwaukee, WI 53201.

IEEE Log Number 8926731.

I. Introduction

In recent years, there has been a growing interest in the study of software complexity metrics. Attempts have been made to understand how metrics can be used to characterize and improve the quality and the maintainability of software. Since the early pioneering work of Halstead [5] and McCabe [9], numerous studies have addressed various aspects of software metrics. Very few studies have focused on the relationship between different metrics, and between metrics and the software development effort, particularly for real-time software.

In this paper, we discuss the results of an experimental investigation of software metrics for a large amount of software developed in an industrial environment and commercially used in a real-time application. The main issues we pursue are the relationship between various metrics, and more importantly, the connections between metrics and the software development effort.

In Section II, we review the definition of software metrics used in our study. In Section III, we outline the software environment and the characteristics of the software studied here. In Section IV, we explore the relationship between a set of software metrics, and in Section V, we consider the linkage between metrics and the software development effort using different approaches.

II. Software Metrics

The metrics that we study include the frequently used metrics—Halstead's program length, McCabe's control complexity, and the less well-known metric called program bandwidth. The significance of these metrics has been discussed extensively before [5], [11] and will not be repeated here. But we will briefly review them.

Let n_1 and n_2 denote the number of unique operators and operands in a program, respectively. Also, let N_1 and N_2 denote the total number of operator and operand appearances. Halstead's program length N is defined as the sum of N_1 and N_2. Since the computation of N may be nontrivial, Halstead defined the following program length estimator (N_H) for N [5]: $N_H = n_1 \log_2 (n_1) + n_2 \log_2 (n_2)$. The following alternative empirical expression (N_J) was reported to be a more accurate estimate of N in [7]: $N_J = \log_2 (n_1 !) + \log_2 (n_2 !)$.

The widely known McCabe's complexity metric MC is the cyclomatic number of the program control graph and is a measure of the complexity of the program's control structure. It has been shown in [9] that MC can be determined by the simple expression $MC =$ number of decisions $+1$.

The program bandwidth BW is an indicator of the average level of nesting of a program and is defined as $BW = ((i * L(i)))/(\text{number of nodes in the program control graph})$ where $L(i)$ denotes the number of nodes at level i [3], [7]. Thus, a straight line program would have a bandwidth of one. On the other hand, a deeply nested program would have a wide band.

Besides the above-mentioned complexity measures, we also considered the following basic and conceptually simple measures: the total number of lines encountered in the main body of the program (including comments), the number of lines of code, the total number of characters, the number of code characters, and the number of comments and the number of comment characters in a program. Parsers were developed and used to compute the various metrics for Pascal and Fortran programs.