[McD89] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, June, 1989.

[NM89] R. Netzer and B. P. Miller. *Detecting Data Races in Parallel Program Executions.* Technical Report 894, University of Wisconsin-Madison, November 1989.

[YT88] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Tran. on Software Engineering*, 14(10):1499–1511, October 1988.

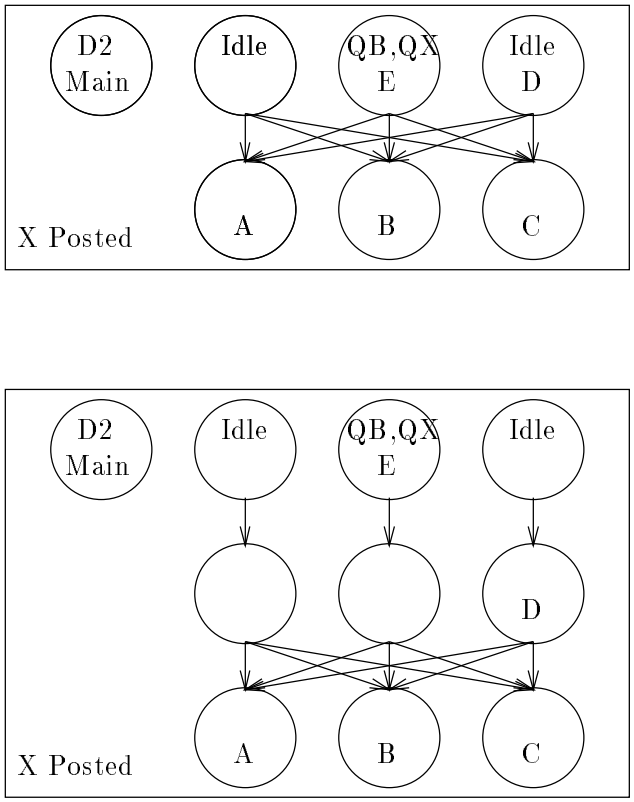

Figure 4.6: Intermediate stages between node 8 and 11

References

[CKS90] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), SIGPLAN Notices*, pages 21–30, March 1990.

[DS90] A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 1990.

[EGP89] P. A. Emrath, S. Ghosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Supercomputing '89*, November 1989. Reno, NV.

[EP88] P. A. Emrath and D. A. Padua. Automatic detection of nondeterminacy in parallel programs. In *Proc. Workshop on Parallel and Distributed Debugging*, pages 89–99, May 1988.

[HL85] D. Helmbold and D. Luckham. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[HMW90] D. P. Helmbold, C. E. McDowell, and J. Z. Wang. Analyzing traces with anonymous synchronization. In *Proc. International Conference on Parallel Processing*, August 1990.

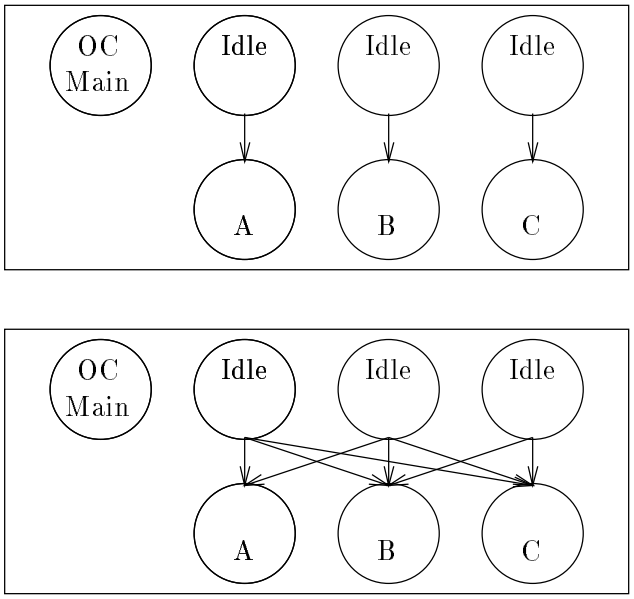

Figure 4.4: SplitNode and MakeCluster applied to node 4 from Figure 4.2.

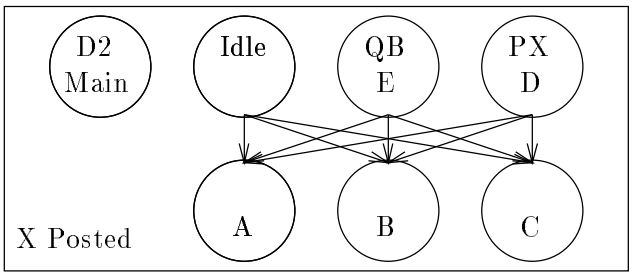

Figure 4.5: Intermediate node between node 6 and 8 before applying AddAction

The transition from node 9 to node 11 is a combination of SplitNode, MakeCluster and AddAction. First, SplitNode and MakeCluster are applied as in the sequence for the transition from node 4 to 5. The AddAction transformation with property 2 is then applied as in the two step transition from node 6 to node 8.

4.5 Identifying task-map subgraphs

The transition from node 8 to node 11 involves a two step process. First an intermediate node is created (the top node in Figure 4.6). Then when node 11 is created from the transition from node 9 it is necessary to recognize that the task-map for the intermediate node is a subgraph of the task-map for node 11. This may in general be a hard problem and work is currently underway to answer the question of just how hard it is. For this example it can be seen that by first expanding the three source nodes (other than Main) node 11 can be obtained by simply adding the edges shown as dotted lines in the node in the bottom of Figure 4.6. The transition from node 10 to node 12 involves a similar subgraph recognition problem.

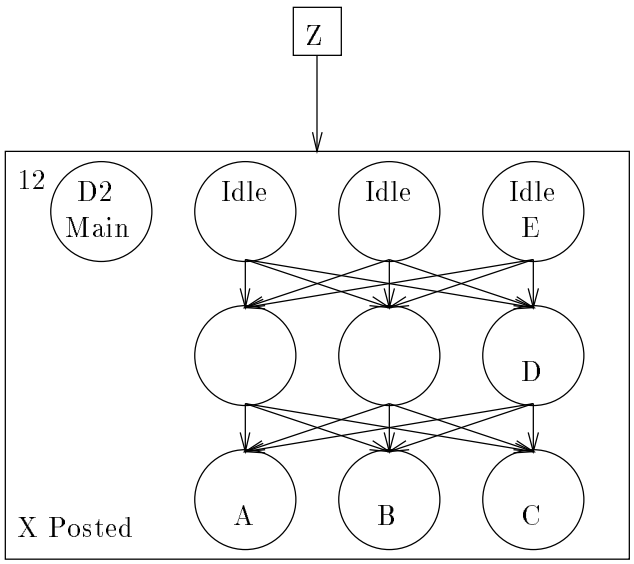

Figure 4.3: Task Compressed Concurrency History Graph (part 2)

example there are three possible source nodes (tasks) that could be selected to be "dispatched". However, if we first apply the SplitNode transformation to the weak cluster containing the three idle nodes then we can get the task-map shown in Figure 4.4. Now transformation MakeCluster can be applied with the result that the three idle nodes form a strong cluster. Given a strong cluster of source nodes it is only necessary to apply the task transition to one of the nodes in the cluster (property 1). This results in node 5 shown in Figure 4.2. The transition from node 5 to node 6 is similar to that from node 4 to node 5 except that the SplitNode is not necessary because the two idle source nodes already form a strong cluster.

4.3 The Post operation and dispatched tasks going idle

The transition from node 5 to node 7 is a simple *Modify* type transition which results in the event X changing to posted. The transitions from node 7 to node 9, from node 8 to node 10 and from node 11 to node 12 are also treated as modify type transitions because a task becoming idle is an "event" that can be detected by other tasks (see "Constraining the task-map" in Section 3.1).

4.4 Applying AddAction

The transition from node 6 to node 8 is created in two steps. The intermediate TC concurrency state shown in Figure 4.5 results from subroutine P posting the event X. From this intermediate TC concurrency state a transition is now enabled that can be handled by an AddAction transformation. In addition, property 2 applies resulting in the intermediate concurrency state being overwritten with the one shown as node 8 in Figure 4.2. The transition from node 7 to node 8 is similar except the first step is the dispatch operation which results in an intermediate TC concurrency state in which an AddAction transformation and property 2 can be applied.

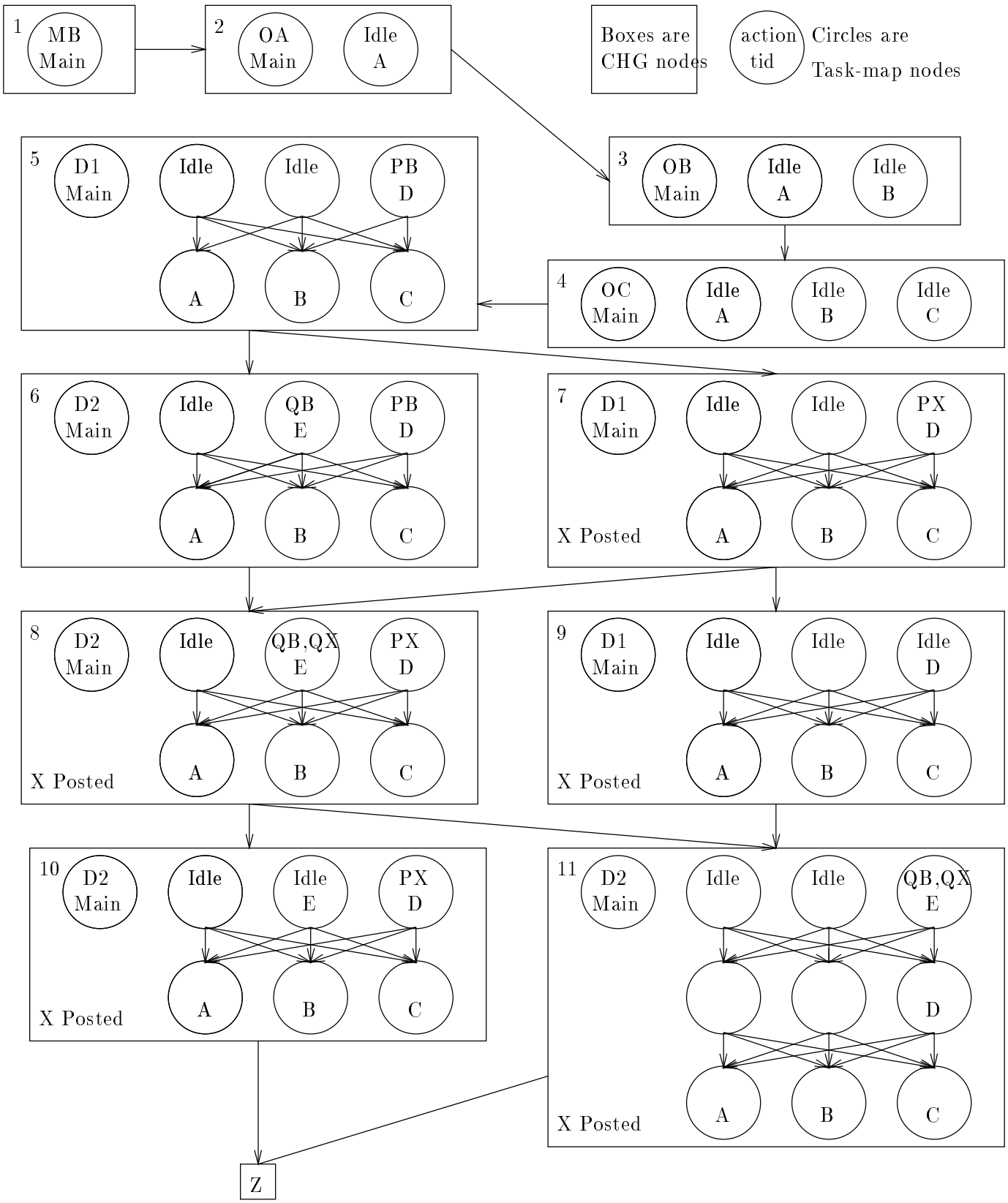

Figure 4.2: Task Compressed Concurrency History Graph (part 1)

Main	**Subroutine P**	**Subroutine Q**
MB: Begin	PB: Begin	QB: Begin
OA: Originate Task A	...	...
OB: Originate Task B	PX: Post X	WX: Wait X
OC: Originate Task C	...	...
D1: Dispatch Any Task D Calling Sub1	End	End
D2: Dispatch Any Task E Calling Sub2		
End		

Figure 4.1: Skeletal Program for Extended Example

4 Extended Example

In this section we present an extended example that uses most of the transformations described previously. The example is shown in Figure 4.1 in a skeletal syntax based on operations found in IBM Parallel Fortran. Short labels are included to allow for concise reference to the program's actions in Figure 4.2. The main task first creates three tasks pointed to by task variables A, B, C respectively. The main task then dispatches two (of the three) tasks to execute subroutines P and Q. The two subroutines synchronize through the use of event X. Once posted, event X remains posted until cleared.

For this example there are two program objects that must be represented, task objects and event objects. For task objects there are two different attributes, the action-set and the task-id as described in Section 2. For event objects there are also two attributes, the name of the event and whether the event is posted or clear.

The two parts of Figure 4.2 show the entire task compressed CHG for the program in Figure 4.1. The outer rectangles correspond to nodes in the TC CHG, and the inner circles correspond to nodes in the task-maps for the corresponding concurrency states. The top half of the inner circles contains the value of the action-set attribute and the bottom half contains the value of the handle attribute. The string "X Posted" in a large box means the event X has been posted. If it is not present, then the event X has not yet been posted.

4.1 The three Originate operations

CHG node 1 in Figure 4.2 shows only one task (source node) in the task-map and the action of the task is "MB" for Main Begin. The first three concurrency state transitions result from *Create* type task transitions ("originate task" has the *Create* characteristic). In each case the action of Main is changed, a new task-map source node is added and the new node is labeled as being in task action "idle" with the specified task variable as a handle.

4.2 The Dispatch operations

The transition from node 4 to node 5 involves a dispatch synchronization operation. As described in Section 3.4, the dispatch synchronization operation has the *Modify* characteristic. Both the main task (having the enabled transition) and the dispatched task have their action-set modified. In this

Finally, node x_1 has its action-set attribute value replaced with $\{a_2\}$. Formally, the set of successor concurrency states is:

$$\{\quad \text{ChangeAction}(\text{AddAttribute}(\text{AddTask}(\text{RemoveAttribute}(M, x_2, (\text{task-id}, t)), x_3), x_3, (\text{task-id}, t)), x_1, a_2) \mid x_1 \in S_{sc}\} \times \{O\}.$$

Wait for a task

In a program, the "wait for t" operation can complete only when the task pointed to by task variable t is idle. This operation has the *T-Read* characteristic. When task transition $a_1 \rightarrow a_2$ represents a "wait for t" operation and is fired in TC concurrency state $\mathcal{C} = (M, O)$, the Constrain transformation on M must be called when generating the successor concurrency states. The set of successor states generated by firing this transition is

$$\{\text{ChangeAction}(M'', x_1, a_2) \mid x_1 \in S_{sc} \text{ and } M'' \in \text{Constrain}(M, \text{idle}, t)\} \times \{O\}.$$

Dispatch a task

The "dispatch any t" operation is an example of task transition which changes the action-set of two source nodes. In addition to changing the action-set for the task executing the dispatch, the action-set of the dispatched task must also be changed and there may also be a new task variable pointing to the dispatched task. Let $\mathcal{C} = (M, O)$ be a TC concurrency state where the transition $a_1 \rightarrow a_2$ for a "dispatch any t" operation is enabled. Let a_s be the first action in the subroutine to which the task is dispatched, S_{idle} be a set of representatives for the source node clusters containing the action "idle", and x be the node in M with a label containing (task-id,t) (if such a node exists). The set of successor concurrency states is

$$\{\quad \text{ChangeAction}(\text{ChangeAction}(\text{AddAttribute}(\text{RemoveAttribute}(M, x, t), x_2, t), x_2, a_s), x_1, a_2) \mid x_1 \in S_{sc}, x_2 \in S_{idle}\} \times \{O\}.$$

A dispatch specific task transition is similar, however it also has the *T-Read* characteristic since the named task must be idle. Therefore the Constrain transformation is applied instead of RemoveAttribute and AddAttribute.

These examples should serve to convince the reader that the transformations on task-maps are general enough to implement the semantics of most (if not all) reasonable synchronization operations.

More complicated transitions may have one or more of the following characteristics:

Modify: The synchronization operation modifies an attribute (in addition to the action-set attribute of the task making the transition).

T-Read: The synchronization operation depends on a task object having a particular combination of attributes.

Create: The synchronization operation creates a new task object.

The characteristics of the transition determine what transformations on the task-map should be performed to avoid invalid successor TC concurrency states. Transitions having the *Modify* characteristic should be implemented with ChangeAction rather than AddAction, and the Constrain transformation should be used when the transformation has the *T-Read* characteristic. The AddTask task-map transformation is designed for those transitions creating new tasks. Although the characteristics of a transition give an idea of which transformations to the task-map are required, the exact sequence of transformations depends on the precise semantics of the synchronization operations involved.

IBM Synchronization Operations

We now describe in detail how the successor TC concurrency states can be generated for a task transition, $a_1 \rightarrow a_2$, involving some of the IBM parallel fortran synchronization operations. As noted in Section 3.3 it often suffices to consider one arbitrary representative of a strong cluster rather than treating each source node in the strong cluster individually. Throughout this section S_{sc} is used to denote a set of source nodes containing one representative of each strong cluster whose nodes are labeled with an action-set containing a_1.

Post an event

An example of an operation that modifies an attribute of a non-task object is the "post" operation. When an enabled task transition $a_1 \rightarrow a_2$ that posts an event is fired in some TC concurrency state $\mathcal{C} = (M, O)$, a set of successor TC concurrency states is created. (If several strong clusters are labeled with action sets containing a_1, then the set S_{sc} will have more than one member and firing the transition results in several successor TC concurrency states.) First, let O' be the non-task portion of $\mathcal{C}$ updated to reflect the posting of the event. Now the set of successor concurrency states is:

$$\{(\text{ChangeAction}(M, v, a_2), O') \mid v \in S_{sc}\}.$$

Originate a task

In a program, the "originate t" operation creates a new task pointed to by task variable t. Assume that the enabled transition from a_1 to a_2 in the TC concurrency state $\mathcal{C} = (M, O)$ involves an originate t operation. The successor concurrency states are created by a series of transformations on M. For each x_1 in S_{sc}, we do the following. First, if some node x_2 has a label containing the attribute-value pair (task-id,t), then that that attribute value pair must be removed from the label on x_2. Then a new task node x_3 not in M must be created and given the label (task-id,t).

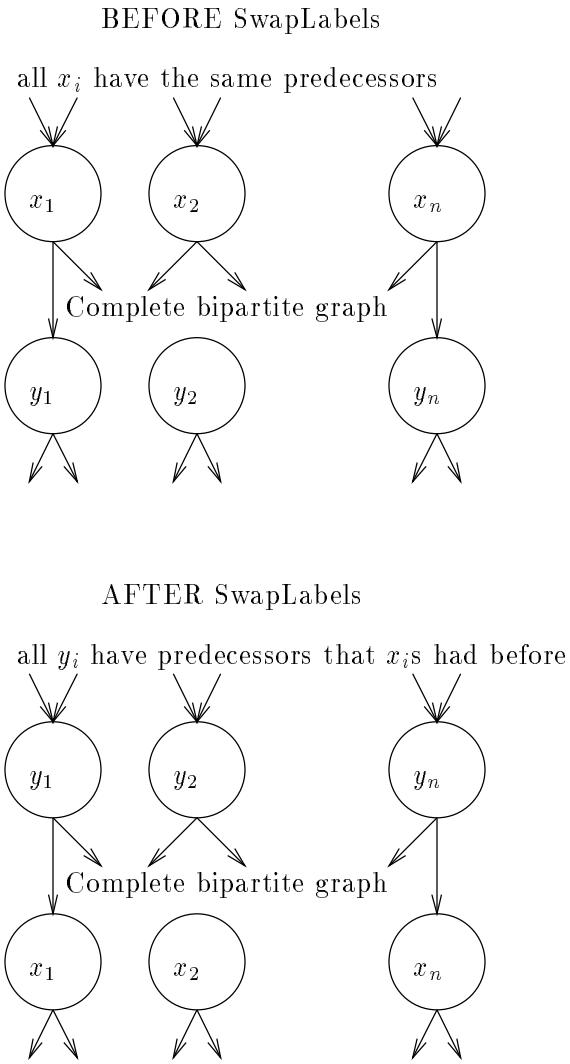

Figure 3.5: A portion of a task-map before and after application of SwapLabels

3.4 Transformation Summary

Each task transition, $a_1 \rightarrow a_2$, requires a slightly different sequence of transformations to the task-map based on the operation(s) represented by the transition. This section describes some of the task transitions that can occur in an IBM parallel fortran program, and how the successor TC concurrency states for these transitions are created. We start by giving a general set of guidelines for generating successor TC concurrency states based on the semantics of operations involved in the task transition.

A **Basic** task transition requires one source node to have an action-set attribute containing the action corresponding to the enabled transition. In addition, a basic task transition may depend on the non-task portion, O, of the TC concurrency state. In this case the successor TC concurrency states are created by applying the AddAction transformation to each cluster labeled with the appropriate action.

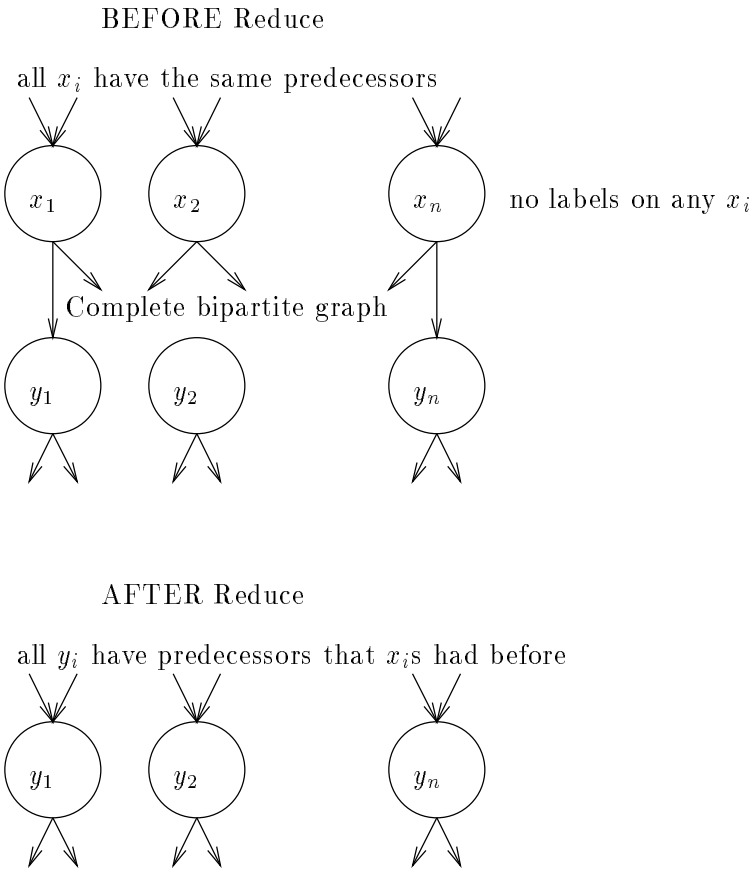

Figure 3.4: A portion of a task-map before and after application of Reduce

Property 1: *Let $\mathcal{C} = (M, O)$ be a TC concurrency state and x_1 and x_2 be source nodes in the same strong cluster of M. The set of successor TC concurrency states generated as a result of enabled task transitions for the task object represented by source node x is equal*[4] *to the set of successor TC concurrency states generated as a result of enabled task transitions for the task object represented by x_2.*

Replacing specific TC CHG nodes with more general nodes

A significant reduction in the number of nodes in a TC CHG can be obtained by utilizing the following property when applying the AddAction transformation. The implication of the property is that when building a TC CHG, whenever a "successor" concurrency state is obtained using AddAction, instead of adding the new node as a successor to the node from which it was generated, the new node *replaces* the node from which it was generated. See Section 4.4 for an example.

Property 2: *If there exist two TC concurrency states $\mathcal{C}$ and $\mathcal{C}'$ where $\mathcal{C}' = AddAction(\mathcal{C}, x_1, a)$ for some x_1 and action a then $ExpandTC(\mathcal{C}) \subseteq ExpandTC(\mathcal{C}')$.*

[4] It is important to note that the nodes in a task-map do not have names. Nodes are distinguished only by their labels (which need not be unique) and their connections to other nodes. We apply names to the nodes in the figures only for expository purposes.

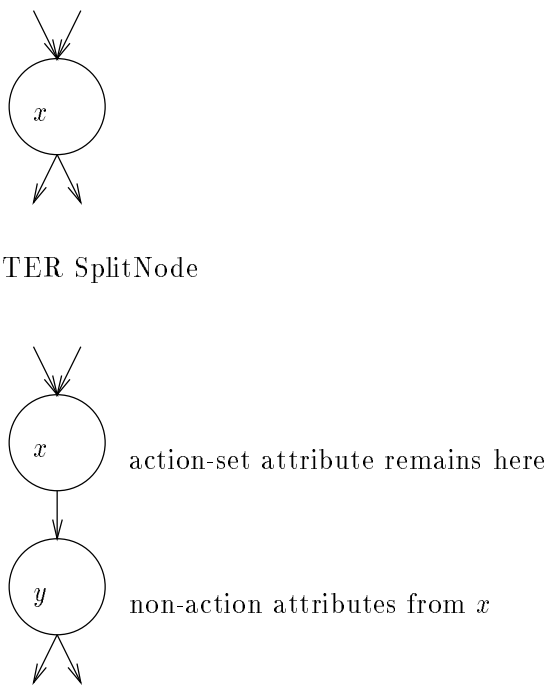

Figure 3.3: A portion of a task-map before and after application of SplitNode

the nodes of S_1 *onto the nodes of* S_2 *SwapLabels(*M*,*S_1*,*S_2*,*f*) yields a new task-map* $M' = (V, E, L')$ *where*

$$L'(v) \quad = \quad \begin{cases} L(v) \text{ if } v \notin S_1 \cup S_2 \\ L(f(v)) \text{ if } v \in S_1 \\ L(f^{-1}v) \text{ if } v \in S_2 \end{cases}$$

$L'(x_1) = L(x_2)$ *and* $L'(x_2) = L(x_1)$.

3.3 Further compressing the task-map

Cluster optimization

Given an initial valid TC concurrency state the complete TC CHG is constructed by repeatedly finding an enabled task transition and applying the necessary transformations to fire the transition. An important observation is that if a set of source nodes form a strong cluster, then only one of them needs to be considered as a candidate for making a task transition that will result in a new TC concurrency state. Any attempt to generate successor concurrency states from another member of the strong cluster results in the same successor concurrency state that has already been generated. This is formalized in the following property.

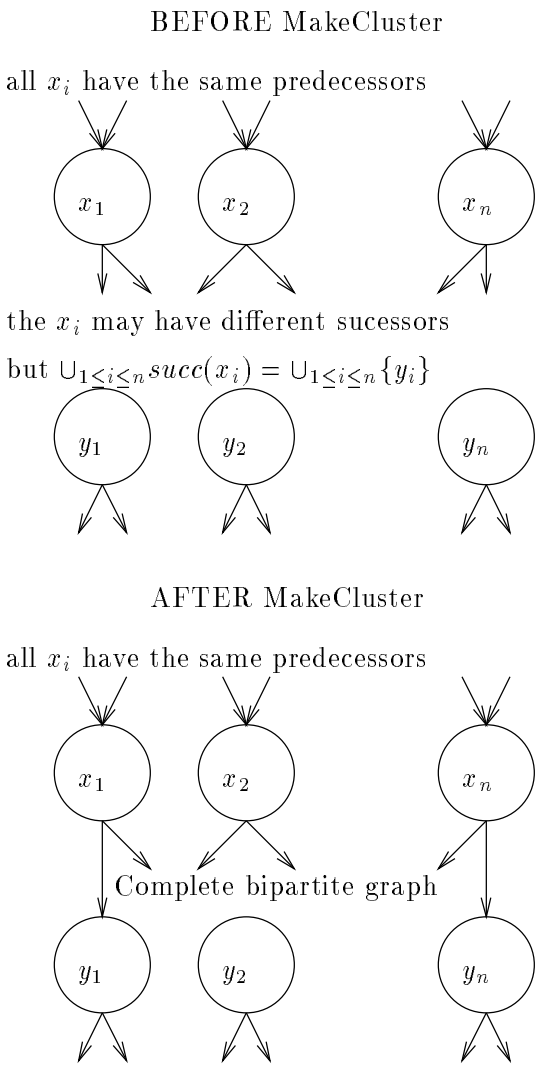

Figure 3.2: A portion of a task-map before and after application of MakeCluster

Transformation 10: (Reduce) *Given a task-map $M = (V, E, L)$ and two weak clusters $S_1 \subseteq V$, $S_2 \subseteq V$, such that for all nodes $v \in S_1$, $L(v) = \emptyset$, and together S_1 and S_2 form a weak component, Reduce(M,S_1,S_2) yields a new task-map $M' = (V', E', L')$ where*

$$\begin{aligned} V' &= V - S_1 \\ E' &= E - \{(x_1, x_2) \mid x_1 \in S_1, x_2 \in S_2\} \\ &\quad -\{(v, x_1) \mid x_1 \in S_1, v \in V\} \\ &\quad +\{(v, x_2) \mid (v, x_1) \in E, x_1 \in S_1, x_2 \in S_2, v \in V\} \\ L' &= L|_{V'}. \end{aligned}$$

Transformation 11: (SwapLabels) *Given a task-map $M = (V, E, L)$, two weak clusters $S_1 \subseteq V$, $S_2 \subseteq V$ of the same size that together form a weak component, and a 1-1 correspondence f from*

Creating new tasks (source nodes in the task-map)

In many parallel programming languages, it is possible to dynamically create tasks. If the enabled task transition invokes the action of creating a new task, then the following transformation is used to reflect this in the task-map. The AddTask transformation will always be used in conjuction with one or more other transformations to reflect the complete semantics of the "task creation" operation (see Section 3.4).

Transformation 7: (AddTask) *Given a task-map $M = (V, E, L)$ and a node $x \notin V$, AddTask(M,x) returns a new task-map $M' = (V \cup \{x\}, E, L')$ where $L'(v) = L(v)$ for all $v \in V$ and $L'(x) =$* (action-set, {idle}).

3.2 Reshaping the task-map

The following transformations can be applied to any legal task-map and the result will be a legal task-map. Furthermore, applying the following four transformations to a TC concurrency state does not affect the expansion of that TC concurrency state.

The MakeCluster transformation enhances the symmetry in a task-map by adding edges as shown in Figure 3.2. The SplitNode transformation creates symmetry by moving labels off of nodes (see Figure 3.3). The Reduce transformation is illustrated in Figure 3.4. This is used to simplify a task-map by eliminating an unnecessary level of unlabeled nodes in the DAG. The SwapLabels transformation is illustrated in Figure 3.5. This is used to change the location of labels in the task-map. Its primary use is in preparing the task-map for a constrain transformation.

Transformation 8: (MakeCluster) *Given a task-map $M = (V, E, L)$ and a set of nodes $S \subseteq V$ such that the nodes in S have the same immediate predecessors and labels, and*

$$|\bigcup_{v \in S} succ(v)| \quad = \quad |S|$$

MakeCluster(M,S) yields a new task-map $M' = (V, E', L)$ where

$$E' \quad = \quad E \cup \{(x_1, x_2) \mid x_1 \in S \text{ and } x_2 \in \bigcup_{v \in S} succ(v)\}$$

Transformation 9: (SplitNode) *Given a task-map $M = (V, E, L)$, a node $x_1 \in V$, and a new node $x_2 \notin V$, SplitNode(M,x_1,x_2) yields a new task-map $M' = (V', E', L')$ where*

$$\begin{aligned} V' &= V \cup \{x_2\} \\ E' &= E - \{(x_1, v) \mid (x_1, v) \in E\} + \{(x_2, v) \mid (x_1, v) \in E\} + \{(x_1, x_2)\} \\ L'(v) &= \begin{cases} \text{actions in } L(v) \text{ if } v = x_1 \\ \text{non-actions in } L(v) \text{ if } v = x_2 \\ L(v) \text{ otherwise} \end{cases} \end{aligned}$$

Our goal when simulating the successful completion of the "wait for task t" operation is to ensure that task t must be idle. If x_1 and x_2 are the nodes as above, and if P is a path from x_1 to x_2 then the goal is accomplished by:

- deleting all edges into P except for edges into x_1,
- deleting all edges out of P except for edges out of x_2, and
- removing all non-idle actions from l_1.

If there are several different paths from x_1 to x_2, or several pairs of nodes that satisfy the conditions on nodes x_1 and x_2, then a new task-map is created for each path, and the one task transition can lead to several successor CHG states.

Given a path, the Psplit transformation eliminates all edges incident to/from the nodes on the path except for: the edges on the path; the edges into the first node of the path; and the edges out of the last node of the path. This transformation ensures that the labels on different path nodes are always mapped to the same task.

Transformation 3: (Psplit)
Given a task-map $M = (V, E, L)$ and a path $P = x_1, x_2, \ldots x_n$ in M, the transformation Psplit(M,P) yields a new task-map $M' = (V, E', L)$ where

$$E' = E - \bigcup_{2 \le i \le n} \{(v, x_i) \mid v \neq x_{i-1}\} - \bigcup_{1 \le i \le n-1} \{(x_i, v) \mid v \neq x_{i+1}\}$$

Psplit is used by the transformation Constrain, which is given only a task-map and two labels which must be mapped to the same task. The Constrain transformation outputs the set of task-maps produced by calling Psplit(M,P) for each path P whose end points contain the labels l_1 and l_2.

Transformation 4: (Constrain) *Given a task-map M and a pair of labels l_1, l_2, the constrain transformation and produces a set of task-maps.*

for each pair $x_1, x_2 \in V$ where $l_1 \in L(x_1)$ and $l_2 \in L(x_2)$ do
    for each path P from x_1 to x_2 in M do
        add the result of Psplit(M,P) to the set of task-maps;
    end for;
end for;

Changing non-action attributes

If the semantics of the enabled task transition cause an attribute to be added or changed then the following two transformations can be used to reflect this in the task-map. It will in general be necessary to combine these transformations with one or more of the previous transformations (i.e., AddAction, ChangeAction and Constrain). Section 3.4 below summarizes how the transformations are combined under various conditions.

Transformation 5: (AddAttribute) *Given a task-map M, a node x and an attribute-value pair α, AddAttribute(M,x,α) returns a new task-map with α added to the label for x.*

Transformation 6: (RemoveAttribute) *Given a task-map M, a node x and an attribute-value pair α, RemoveAttribute(M,x,α) returns a new task-map with α removed from the label for x.*

Adding an action to the action-set

If an enabled task transition $a_1 \rightarrow a_2$ does not change any attributes and does not depend upon any task attributes[2] then the new task action, a_2, can be added to the action-set attribute for every source node labeled with an action-set containing a_1. This is specified by the transformation AddAction.

Transformation 1: (AddAction) *Given a task-map $M = (V, E, L)$, a source node $x \in V$ and an action a, AddAction(M,x,a) returns the a task-map with action a added to the action-set of every node in the strong cluster containing x.*

Changing the action-set attribute

If the enabled task transition $a_1 \rightarrow a_2$ modifies some attribute (in addition to the the action-set attribute) and is not dependent upon any task attributes, then the new concurrency state is created by taking a source node whose action-set attribute contains a_1 and replacing its action-set attribute with the set containing *only* the new action, a_2. This is specified by the transformation ChangeAction.

Transformation 2: (ChangeAction) *Given a task-map M, a source node x and an action a, ChangeAction(M,x,a) returns a new task-map where $\{a\}$ is the action-set labeling x.*

Constraining the task-map

The idea behind task-maps is to represent many concrete concurrency states with a single TC concurrency state. However, this means that we will occasionally encounter a TC concurrency state $\mathcal{C}$, where a task transition is enabled in only some of the concrete concurrency states in ExpandTC($\mathcal{C}$). When firing these task transitions we must ensure that the resulting TC concurrency states are valid. This means that we must first create a restricted TC concurrency state (or set of states) whose expansion is the subset of ExpandTC($\mathcal{C}$) where the transition is enabled. The Constrain transformation is designed to create task-maps for these restricted TC concurrency states.

The Constrain transformation is applied when a transition is enabled only if some task has a certain combination of attributes. For example, the operation "wait for task t" can only complete when task t is idle. In a TC concurrency state, task t can be idle if and only if there is a path in the task-map containing nodes[3] x_1 and x_2 (labeled with l_1 and l_2 respectively) where:

- l_1 includes an action-set attribute containing the action "idle," and
- l_2 includes the attribute-value pair (task-id,t).

Furthermore, task t must be idle in this TC concurrency state if every full path containing a node labeled with an attribute-value pair (task-id,t) also contains a node labeled with the action-set {idle}.

[2] All task transitions cause (at least potentially) the action-set attribute of one task object to be modified. I.e. the action-set containing the tail of the enabled task transition. In the following discussion this will be assumed so that the statement "no attributes are modified" should be read as "no attributes other than the action-set attribute containing the head of the enabled transition are modified."

[3] It might be the case that $x_1 = x_2$.

For any node $x \in V$, when the task-map is clear from context, we let

- pred(x) be the immediate predecessors of x, i.e. the set $\{v : (v, x) \in E\}$;
- succ(x) be the immediate successors of x, i.e. the set $\{v : (x, v) \in E\}$;

Our compression techniques revolve around symmetries between the tasks. Therefore it is important to identify those portions of the task-map graph that are similar.

Definition 7: *Two task-map nodes x_1 and x_2 are* weakly equivalent *if*

1. *pred(x_1) = pred(x_2) and*
2. *succ(x_1) = succ(x_2).*

Definition 8: *Two task-map nodes x_1 and x_2 are* strongly equivalent *if*

1. *they are weakly equivalent and*
2. *$L(x_1) = L(x_2)$.*

Definition 9: *A* weak cluster *is an equivalence class with respect to the relation* weakly equivalent *for task-map nodes.*

Definition 10: *A* strong cluster *is an equivalence class with respect to the relation* strongly equivalent *for task-map nodes.*

Strong clusters are very important, as it often suffices to consider a single representative of a strong cluster when transforming the task-map (see Section 3.3). This saves the work (and additional CHG states) that would result from transforming each node of the strong cluster in turn.

Definition 11: *A* weak component *in a task map consists of two weak clusters S_1 and S_2 with the property that for each $x_1 \in S_1$ and $x_2 \in S_2$, the edge (x_1, x_2) is in E.*

Definition 12: *A* strong component *in a task map consists of two strong clusters S_1 and S_2 with the property that for each $x_1 \in S_1$ and $x_2 \in S_2$, the edge (x_1, x_2) is in E.*

3.1 Generating Task Compressed Concurrency States

To build a task-compressed CHG, an initial task-compressed concurrency state is specified and then new concurrency states (nodes) are added to the CHG by firing task transitions that are enabled. The set of nodes in the final task-compressed CHG is the smallest set of concurrency states which contains the initial task-compressed concurrency state and is closed under the "fire-enabled-transition" operation.

When a transition is fired, certain transformations must be applied to the TC concurrency state in order to generate the new TC concurrency state. The choice of transformations depends on the semantics of the enabled task transition.

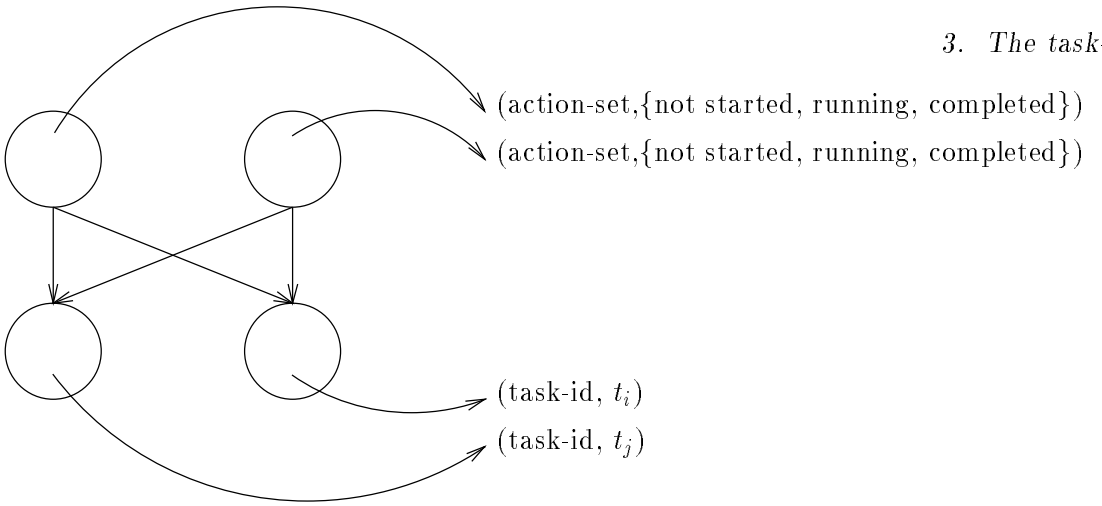

Figure 3.1: Task-map for the only node in the TC CHG corresponding to example from Figure 1.1.

one action-set attribute, we adopt the convention that *every source node is labeled with an action-set attribute and non-source nodes never contain action-set attributes.* This makes it possible to establish a correspondence between the source nodes and the task objects represented by the task-map. It also emphasizes the special nature of the action-set attributes.

The following definition is motivated by the intent that a well-formed task-map indicates how the attributes can be partitioned among the tasks.

Definition 6: *A task-map M is* legal *if whenever S is a maximal set of node-disjoint full paths in M, then every node in M is on one of the paths in S.*

In a legal task-map, every node (and hence every label) is used once in every maximal set of node-disjoint full paths. Legal task-maps also have a structure that simplifies their analysis.

Lemma 1: *If M is a legal task-map then every full path in M is in some complete set of paths for M.*

Lemma 2: *If M is a legal task-map then a complete set of paths in M can be constructed in time proportional to the number of arcs in M.*

From the practical point of view, the race conditions in a task compressed concurrency state described by a legal task-map can be easily detected (in either $O(v+l)$ or $O(l \log l)$ time where v is the number of variables in the program and l is the total length of the read/write lists for that compressed concurrency state). Furthermore, given a task compressed concurrency state described by a legal task-map, it is easy to compute the successor task compressed concurrency states. In the remainder of this paper, we assume that all task-maps are legal.

Before describing how task transition firing transforms task-compressed concurrency states we first need some more definitions.

Recall that $M = (V, E, L)$ is a task-map where:

$$\begin{aligned} V &= \text{the set of nodes in } M. \\ E &= \text{the set of arcs in } M. \\ L &= \text{the labeling function such that for } x \in V, \\ &\quad L(x) = \text{the set of attribute-value pairs associated with } x \end{aligned}$$

way to form the attribute lists corresponding to the non-task objects. Formally a TC concurrency state, $\mathcal{C}$, is represented as a pair (M, O) where:

M is a task-map indicating the possible grouping of attribute-value pairs into attribute lists corresponding to task objects, and

O is an association of attribute-value pairs into attribute lists corresponding to the non-task objects.

An arbitrary task-map will generally be written $M = (V, E, L)$ where V is the set of nodes in the DAG, E is the set of arcs in the DAG, and L is a labeling function on the nodes, indicating the attribute value (or set of attribute values) labeling each node.

Definition 3: *A* full path *in a task-map is a path from a source node (indegree zero) to a sink node (outdegree zero).*

A complete set of paths *in a task-map is a set of node-disjoint full paths where every source is on one of the paths.*

Any path in a task-map can be viewed as an attribute list by concatenating the labels on the path. The method of associating attributes with objects in a TC concurrency state $\mathcal{C} = (M, O)$ is to use the associations in O for the non-task objects plus, for any complete set of paths in the task-map, the task attribute lists corresponding to the paths in the set. Note that this mapping is highly nondeterministic and that different complete sets of paths will generally yield different concrete concurrency states. We use the following definition to denote the relationship between concrete concurrency states and TC concurrency states.

Definition 4: *Given a task compressed concurrency state $\mathcal{C}$, the* expansion *of $\mathcal{C}$ (often written* ExpandTC*($\mathcal{C}$)) is the set of all concrete concurrency states represented by $\mathcal{C}$.*

Algorithm 1: *ExpandTC($\mathcal{C}$=(M,O)) is computed by the following:*

for each complete set of paths in M do
    let S be the set of attribute lists associated with the paths;
    for each way of selecting one action from each of the action-set attributes do
        form $\mathcal{C}'$ from $S \cup O$ by deleting all but the selected action from each action-set;
        add the concrete concurrency state $\mathcal{C}'$ to ExpandTC($\mathcal{C}$);
    end for;
end for;

The following definition is used to determine what are meaningful TC concurrency states.

Definition 5: *A TC concurrency state, $\mathcal{C}$ is* valid *if every concrete concurrency state in ExpandTC($\mathcal{C}$) is also in the program's concrete CHG.*

As an example, the concrete CHG in Figure 1.1 can be represented by a task compressed CHG containing only a single node (concurrency state). The task-map for this one task compressed concurrency state is shown in Figure 3.1. The task compressed CHG is valid because every concrete concurrency state in its expansion is also in the CHG in Figure 1.1.

In a task-map, the number of sources (nodes of indegree zero) equals the number sinks (nodes of outdegree zero) which equals the number of task objects represented. Since each task has exactly

storage for the CHG may be reduced. The main result of this paper is a general concurrency state representation which facilitates the construction of the CHG. The primary objective is to minimize the total storage required for CHGs built out of generalized concurrency states.

As much as possible the semantics of task transitions and specific attributes will be left unspecified, thereby maintaining as much language independence as possible. One attribute that will be found in all representations of concurrency states is the *action-set* attribute for task objects. The value of an action-set attribute is a set of actions, where each action in the set identifies some point in the program that can be reached by a single task (thread). The meaning of the attribute is that the associated task object can be in *any* of the actions in the set[1].

One additional attribute that will be used frequently in examples in this paper is the *task-id* attribute. The value of a task-id attribute is the name of a program variable that currently "points" to the associated task. For example if an attribute list for a task object contains a task-id attribute with value TV, then the program variable TV currently points to the task defined by that attribute list.

To build a concrete CHG, an initial concrete concurrency state is specified and then new concurrency states (nodes) are added to the CHG by "firing" task transitions that are "enabled". The set of nodes in the final concrete CHG is the smallest set of concurrency states which contains the initial concrete concurrency state and is closed under the "fire-enabled-transition" operation.

To build a task-compressed CHG, an initial task-compressed concurrency state is specified and then new concurrency states (nodes) are added to the CHG by firing task transitions that are enabled. The set of nodes in the final task-compressed CHG is the smallest set of concurrency states which contains the initial task-compressed concurrency state and is closed under the "fire-enabled-transition" operation.

Definition 1: *A task transition $a_i \rightarrow a_j$ is* enabled *for a task object t in concurrency state $\mathcal{C}$ if:*

1. *task object t has an action-set attribute containing a_i,*
2. *there is a possible transition in the program from action a_i to action a_j, and*
3. *any preconditions necessary for the task to execute sequentially from action a_i to action a_j are satisfied by the other information in $\mathcal{C}$.*

We also say that task t is enabled *when these conditions are satisfied.*

Definition 2: *An enabled task transition is* fired, *resulting in a new concurrency state by adding, deleting or modifying the attribute-value pairs, the set of objects and/or the mapping functions of the concurrency state containing the enabled task transition, as specified by the semantics of the operations in the task transition.*

3 The task-map

In this section we describe a generalized concurrency state representation called a *task compressed concurrency state* or TC concurrency state. A TC concurrency state usually contains several ways of mapping the task attribute-value pairs to the task objects as indicated by a labeled DAG called the task-map (described below). As in concrete concurrency states, there is only one

[1] For concrete concurrency states these sets always have cardinality one.

task attributes	
attribute	value range
last synchronization operation completed	program statement
program task variable indicating *this* task	any task variable
event attributes	
attribute	value range
wait count	integer
post count	integer
posted or waited on count	integer
lock attributes	
attribute	value range
status	set or clear

Table 2.1: Some attributes and their value ranges

would be a snapshot of the entire program state) and a CHG with far too many nodes. However, if we could create such a CHG we could accurately report exactly those races that occur in some execution of the program. A major problem is finding a good compromise between the detail in a concurrency state and the accuracy of the generated race warnings. Being very careful about how resolution is given up is the key to solving this problem. (For example, consider a TV image. One way to give up resolution is to use fewer pixels, another is to use fewer colors.)

In Section 2 we describe the basic representations of a concurrency history graph. In Section 3 we present a variation of the basic concurrency history graph that requires significantly (possibly orders of magnitude) less storage for the concurrency history graph. Finally in Section 4 we present an extended example that demonstrates the techniques described in Section 3.

2 Representing Concurrency States

A concurrency state is a snapshot of the execution of a parallel program. Different representations of concurrency states may store the information from the snapshot at different resolutions. We will represent a concurrency state as a set of attribute lists composed of attribute-value pairs. Each attribute list will represent a different program object. These objects could include tasks, events, and locks. Some possible attributes and their value ranges are shown in table 2.1. We will refer to this representation of a concurrency state as a *concrete concurrency state* (to distinguish it from the other representations introduced below). A CHG in which the nodes are concrete concurrency states is called a concrete CHG.

A more general representation of a concurrency state is as a collection of attribute-value pairs, a (possibly implicit) set of objects, and a set of functions mapping the attribute-value pairs to objects. If there is only one function in the set, then each of these more general concurrency states represents a single concrete concurrency state. When multiple functions are present a single generalized concurrency state represents several concrete concurrency states. Therefore a CHG built from generalized concurrency states may contain fewer nodes than one built from concrete concurrency states. If the set of mapping functions can be efficiently described, then the total

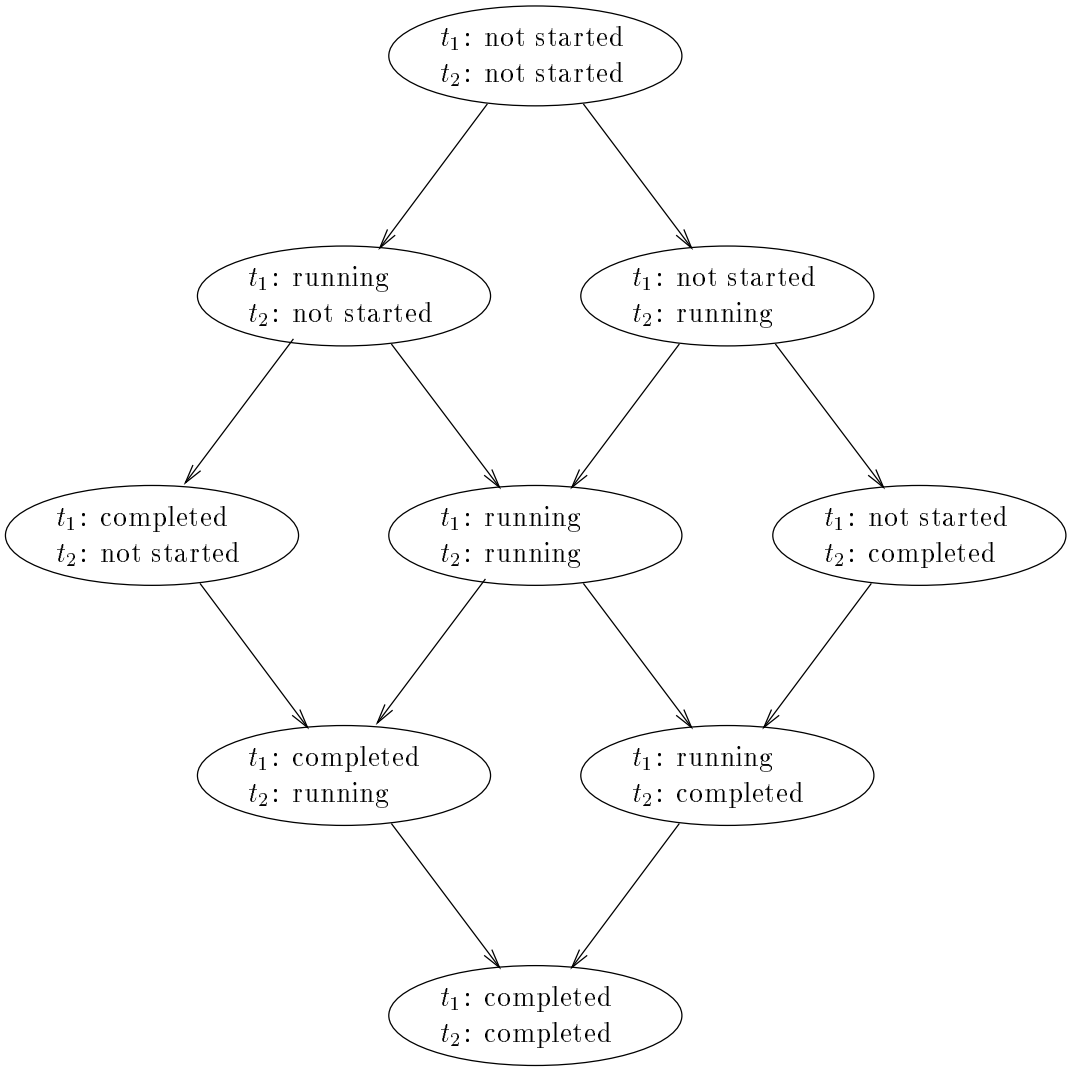

Figure 1.1: Complete CHG for two unsynchronized tasks.

not constrained. By associating shared data accesses with task transitions (edges in the CHG) it is possible to identify both concurrent and order races in the CHG (or the program it represents).

The problem with a low resolution CHG like the one in Figure 1.1 is that it may contain task transitions and concurrency states that cannot occur in any actual execution of the program. This could result in races being reported that could never occur in the program being analyzed. In our simple example, it appears that any pair of accesses by the two tasks could potentially execute concurrently. Therefore, a race will be reported on each access to a shared variable. It might instead be the case that there is additional synchronization, not represented by this low resolution CHG, which prevents the races from occurring.

At the other extreme we can imagine a very high resolution concurrency state where a task transition is the execution of a single machine instruction and the global information includes the entire contents of memory. This would result in huge concurrency states (each concurrency state

1 Introduction

The need to identify access races in parallel programs is a well recognized problem. Current approaches to solving this difficult problem include compile time analysis [CKS90, McD89, YT88], run time detection [HL85, DS90] and post mortem trace analysis [HMW90, EP88, EGP89, NM89]. No single approach has yet been developed that is clearly superior under all conditions. Furthermore, it is our belief that a program development system will include aspects of all three applying them in combination and separately as dictated by the application (and further research).

In this paper we describe a method for detecting races at compile time. The analysis is based on enumerating all possible program states with sufficient resolution to determine which program statements might execute concurrently. These program states are called *concurrency states*, primarily because the resolution of the "program counter" for each of the concurrent tasks represented in a concurrency state is often an explicit synchronization statement in the program (i.e., the program counter will appear to jump from synchronization statement to synchronization statement).

Each concurrency state includes some task specific state information for each concurrent task (or process), and possibly some global information. The granularity of the information contained in a concurrency state will affect the number of possible concurrency states (and the accuracy of error reports derived from the concurrency states). For example, we could ignore the global information and limit the task specific information to two attributes: an identifying name and one of the actions *not-started*, *running*, or *completed*. A concurrency state would then consist of one entry for each task indicating its name and which of the three possible actions it was in. If the tasks were completely unsynchronized, they could be in any of the 3^t (where t is the number of tasks) different combinations of actions. In general, when there is no global information and the each task can be doing any of n different actions, the number of possible concurrency states is $O(n^t)$. In practice the actual number of concurrency states will be less than n^t because task synchronization prevents some combinations of task specific values from occurring.

The brute force enumeration of all possible concurrency states is too expensive to be a practical approach to compile time detection of race conditions. In this paper we will describe how it is possible to avoid enumerating each individual concurrency state and instead enumerate *compressed concurrency states* where each compressed concurrency state represents many of the individual concurrency states.

The (uncompressed) concurrency states can be organized into a graph called a *Concurrency History Graph* (CHG). The nodes in a CHG are the concurrency states. The edges in a CHG represent the *task transitions*. A task transition corresponds to one task proceeding sequentially according to its specification (program) until the task specific action and/or some global state stored in the concurrency state has changed. These task transitions are closely related to the resolution of the concurrency states. In the above example, each task makes two transitions (*task-not-started* $\rightarrow$ *task-running*, followed by *task-running* $\rightarrow$ *task-completed*). Using this simple (i.e., low resolution) concurrency state, the entire CHG for a program with two unsynchronized tasks is given in Figure 1.1.

Access races can be divided into two groups, *concurrent races* and *order races*. In a concurrent race, the two accesses might execute concurrently (i.e., at the same time), whereas in an order race, the two accesses are prevented from executing concurrently but the order in which they execute is

Computing Reachable States of Parallel Programs

David P. Helmbold
Charles E. McDowell

April 23, 1991

Board of Studies in Computer and Information Sciences
University of California at Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

A concurrency history graph is a representation of the reachable states of a parallel program, A new abstraction for representing the state of a parallel program is presented. This new abstraction is more general than previous work by the authors. At the same time, the new abstraction makes it possible to produce concurrency history graphs that require much less storage than that suggested by a simple worst case complexity analysis. Concurrency history graphs based on this new abstraction form the foundation upon which a static analysis tool capable of detecting race conditions in parallel programs is being built.

keywords: parallel processing, debugging, static program analysis