# THE INTERACTION OF THE FORMAL AND THE PRACTICAL IN PARALLEL PROGRAMMING ENVIRONMENT DEVELOPMENT: CODE

John Werth, James C. Browne, Steve Sobek,
T. J. Lee, Peter Newton, and Ravi Jain

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-09 April 1991

# The Interaction of the Formal and the Practical in Parallel Programming Environment Development: CODE

John Werth, James C.Browne, Steve Sobek, T.J.Lee, Peter Newton, Ravi Jain
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
email browne@cs.utexas.edu
512-471-9579

## Abstract

The most visible facet of the Computationally-Oriented Display Environment (CODE) is its graphical interface. However, the most important fact about CODE is that it is a programming system based on a formal unified computation graph model of parallel computation which was intended for actual program development. Most previous programming systems based on formal models of computation have been intended primarily to serve as specification systems. This paper focuses on the interaction between the development of the formal model of parallel computation and the development of a practical programming environment. Basing CODE on a formal model of parallel computation was integral to attainment of the initial project goals of an increase in level of abstraction of representation for parallel program structure and architectural independence. It also led to other significant research directions, such as a calculus of composition for parallel programs, and has suggested other directions of research in parallel programming that we have not yet had the opportunity to pursue. We hope this experience with the interaction of the theoretical and the practical may be of interest and benefit to other designers and developers of parallel programming systems.

## I. Introduction

### I.1. Parallel program development: The CODE/ROPE design environment

The initial version of CODE (the Computationally-Oriented Display Environment) [BRO89] was based upon a unified model of parallel computation defined by Browne [BRO85,BRO86] and extended to have a proper formal basis by Sobek [SOB90]. The original motivations for basing the CODE programming system on a formal model of computation were to be able to raise the level of abstraction at which parallel programs are expressed and to provide a solid foundation for the difficult problem of compiling to multiple parallel architectures. As is frequently the case in research, unplanned results arose. CODE requires, because of the level of granularity of its typical unit of computation, a component library. ROPE (the Reuseability-Oriented Parallel Environment), which implements a library capability for CODE was a response to this requirement. ROPE stores and retrieves subgraphs for insertion in CODE graphical representations of parallel programs. Observation and analysis of the process of inserting subgraphs into existing program graphs led to the observation that there was a well-defined calculus of composition for composing subgraphs representing parallel computation structures defined in the CODE representation into "larger" graphs also representing parallel computation structures. Continuation of this line of reasoning revealed that there is a full calculus (or algebra for that matter as well) of composition for parallel program structures based on the definition of program elements in CODE and ROPE [BRO89]. Characterization of the family of program graphs has also suggested some algorithms for identification of parallelism in existing programs.

It was actually the case that the first implementation of CODE preceded the formalization of the model of parallel computation. The initial implementation has also undergone substantial revision

as a result of deficiencies revealed by use. The implementation and the formal model were brought back into closer harmony in the current version of CODE , Version 1.2.

This paper first defines and illustrates the formal model of parallel computation underlying Version 1.2 of CODE. There are few if any parallel programming languages intended for production use of executable programs for which a precise formal model can be written down. The formally based parallel programming systems such as CSP [Hoa78] and UNITY [CHA88] have not typically not been given full implementations although there are exceptions such as Linda [GEL85]. Consistency between CODE and its model of parallel computation is possible because CODE expresses only the parallel structure of a program and because of the high level of declarative abstraction at which CODE expresses parallel structure.

Use of the formal model of parallel structure as the basis for CODE has strongly influenced its evolution. Extensions to provide greater convenience in expressing programs has been constrained to constructs which can be straightforwardly incorporated in the model of parallel computation. This has enabled us to avoid introducing inconsistent or incompletely thought-through constructs on several occasions, notably in the representation of exclusion relations. There are possibilities for analysis of safety, liveness and performance properties of executable parallel programs based on this formal model which have not yet been exploited.

In the final section we discuss the calculus of composition in this model of parallel computation. The calculus of composition has significant applications in the context of parallel programming environments or at least for CODE/ROPE. Expressions in the calculus of composition provide a compact representation for complex graph structures. The calculus of composition is the natural text-based equivalent to the graphical syntax currently used by CODE. There may be many occasions where a mixed-basis representation of a parallel program will be more readable and compact than any single basis representation. The operators provide a powerful shorthand notation for specification of the connections to instantiate a subgraph in a program graph. They are the basis for in-depth type checking for manual subgraph insertions. They also are a basis for automatic composition of parallel programs which will be correct provided the components are correct. The operators can also be used to specify connectivity properties required of a subgraph.

## II.2. The Calculus of Composition - Software Engineering Concepts and Related Research

The calculus of composition for parallel programs for CODE integrates the concepts of hierarchy and interconnection or interface specification. The concepts of hierarchy [PAR72] and interconnection/interface specification [GOG86, PRI82] are foundations of modern software engineering. Top-down design and reuse of software components for sequential language based programming systems are founded on these two concepts. There have been proposals for composition operators for parallel computation structures [PRA82] and parallel programs [CHA89]. Brock and Ackerman have given composition operators for dataflow parallel structures [BRO81]. Chandy and Taylor [CHA89] have applied the UNITY composition operators in a form for directly expressing executable programs in PCN. Adler [ADL88] recently defined an algebra for data flow diagram decomposition targeting support for top down design of programs.

The calculus of composition and decomposition given herein support both top-down design of parallel programs and reuse of software components (bottom-up design). An arbitrarily complex parallel program structure can be derived top down by expressing a single node as an equivalent graph defined in terms of the composition operators, expressing the nodes of that graph in terms of an equivalent graph defined in terms of the composition operators, etc. Software component reuse is supported by constraining the top down process to terminate on existing components, by allowing components to be composed into new components which meet the specifications for a particular instantiation and by providing a basis for automatic selection and instantiation of components in parallel computation structures.

## II. An Informal Introduction to CODE/ROPE

CODE is a program development system for parallel programs. In CODE, programs are organized as graphs with three possible types of nodes and two possible types of arcs. The nodes are associated with computations, and the arcs are associated with data.

1. **Directed arcs** (denoted by arrows, also called data dependencies) indicate data being generated by the source node, and then flowing to the sink node.
2. **Hyperarcs** (undirected arcs potentially joining more than two nodes, denoted by dotted lines and also called exclusion dependencies) indicate data which is shared by the computations represented by the nodes joined by the hyperarc. The hyperarcs have associated constraints that control access to the data; this is the basic mechanism for preventing race conditions.
3. **Schedulable Units of Computation (SUC)** nodes (denoted by circles) are associated with some computation. They are distinguished by only being able to execute when data is present on all incoming directed arcs. They place data on all of their outgoing directed arcs at the end of their execution.
4. **Switch nodes** (denoted by diamonds) perform specialized computations associated with making choices as well as merging and distributing data. They are enabled for execution if data is present on any input arc. These nodes may also place data on any subset of their outgoing directed arcs after execution.
5. **Subgraph nodes** (denoted by boxes) encapsulate computations performed by entire graphs.

Program development with CODE requires first specifying the graph (which represents the overall organization of the computation) and then providing details about each graph element:

For SUC nodes,the user supplies a computation in the form of a subprogram which may come from a library or be written from scratch.

For switch nodes, the user supplies a condition on each input arc which describes the conditions under which data on that arc are to be passed through the node and on to the destination node(s) to which it is routed.

For directed arcs, the user supplies a data name and data type.

For hyperarcs, the user supplies a data name, a data type and a data sharing constraint to be preserved by the system among the nodes sharing the data . (We will seelater that this condition is actually specified by annotating the nodes)

Once program development is complete, the user is able to request translation of the program to any of several executable forms. Each executable form is targeted to the specific hardware and software environment in which the program will be executed. By analogy with the compilation process, the system specific portion of CODE which creates these executables is called a backend. The current version of the software supports backends for Ada and Fortran on a variety of architectures.

ROPE is a software reuse system integrated with CODE [LEE90]. The user model is essentially one of selecting a subgraph from a library and then connecting that subgraph to the CODE graph under construction by using data dependencies and exclusion dependencies. The user may also create new modules and insert them in the library.

The implementation of CODE/ROPE is distinguished by the software engineering features which have been incorporated to facilitate its practical use. These features encompass the user interface, provisions for reuse of program fragments, and facilities for structuring programs. The system has been through several versions and has had substantial use by graduate and undergraduate students in classes. A list of CODE related documents appears in the bibliography.

## II. A Formal Definition of the Unified Computation Graph Model

The purpose of this section is to give a formal notation for the model of computation used in the definition of Versions 1.0 and 1.2 of the CODE/ROPE systems developed at the University of Texas at Austin. Though these definitions differ to some degree from those of [Sobek90], closely related material may be found there. The approach taken here is fairly general and in some cases contains ideas not actually implemented in CODE 1.2. Notes are supplied to indicate limitations of the existing CODE/ROPE system. A new version, CODE/ROPE 2.0, is currently under development; it is based on a further enriched model of parallel computation with complete realization of the model of computation [Newton91].

The formal definition is organized as follows.

1. The elements of the unified model of parallel computation are defined in set notation in terms of a graph model, the **Unified Computation Graph (UCG)** model (Section II).
2. The semantics of each element of the model are specified as they are realized in CODE 1.2 (Section III).
3. The graph model is shown to lead naturally to a formal definition of a reuseable component as any full closed subgraph of a **UCG** (Section IV).

Section II is necessarily quite formal. While the formality is essential to completeness, the concepts are simple and what is essential for extracting the content of the rest of the paper can be obtained by reading the definitions and the explanatory notes, the italicized intuitive notes and Section III.

**Definition II.1. Type System**

Given a countably infinite set of values U, a type system is a family of subsets of U called types. There are n distinguished finite subsets B1, .., Bn of U called **basic types**. A **type, t,** is either

1. A basic type
2. U
3. (Arrays) If I1, ..., Im are finite totally ordered sets and t is a type then the set of functions f: I1× I2×...×Im -> t is a type.

**Definition II.2.** A **Unified Computation Graph (UCG)**, G, is a tuple (S, D, E, T)

S is the set of **nodes,** s, of G.
D is the set of **data dependencies**, d, of G.
E is the set of **exclusion dependencies**, e, of G.
T is some **type system**

We use an operator style notation; for example, if G is a UCG then S(G) is the set of nodes, D(G) is the set of data dependencies of G. We write n:t if identifier n has type t.

**Definition II.3.**

A **Data dependency d** has an associated pair of nodes, s1 = Source(d), s2 = Sink(d), a type t = T(d), and a name N(d).
An **Input data dependency** d is one with source(d) = sI; that is. d= ( (sI, s), n:t)
An **Output data dependency** d is one with sink(d) = sO; that is. d= ( (s, sO), n:t)

*Intuitively a data dependency is a buffer which carries a sequence of typed values from source to sink.*

**Notation II.1.**

a. If d is a data dependency then
   S(d) = {Source(d), Sink(d)}, the node set of the dependency

b. If s is a node then
   In(s) = {d | Sink(d) = s)}, Out(s) = {d | Source(d) = s)}
   D(s) = In(s) union Out(s) , E(s) = {e | s ∈ S(e) }

c. If G is a UCG then
   ID(G) is the set of input dependencies contained in D(G)
   OD(G) is the set of output dependencies contained in D(G)
   IOD(G) = ID(G) ∪ OD(G) is called the **I/O dependencies** of G
   IND(G) = D(G) - IOD(G) is called the **internal dependencies** of G

There are unique start (sI) and end (sO) nodes in S(G) and G has the property that for every node s there is a directed path from sI to sO passing through s. There is a unique data dependency du of type U from sI to sO. The trivial UCG for type system T is ({sI,sO}, {du}, { },T). We assume there are no dependencies with sink sO and source sI except du.

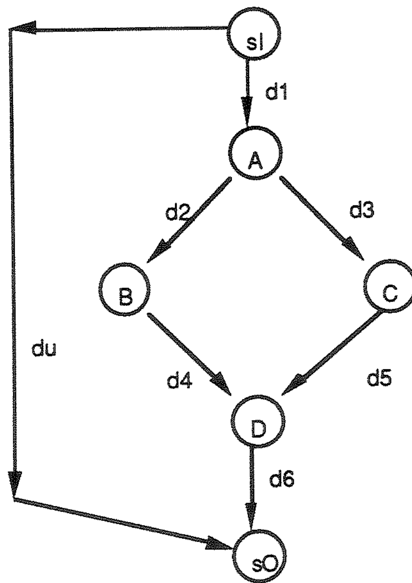



Figure 1
A typical UCG

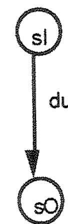



Figure 2
The Trivial UCG

*Figure 1 might be a program which generates a vector of integers in A, while B and C sort the list into sublists of even and odd integers and D prints the two lists. d2, d3, d4 and d5 are data dependencies each of which carries a vector of integers. sI, sO, d1, d6 and du, while they are not essential to the specification of a computation are necessary to the definition of reuseable components for the UCG model.*

## CODE 1.2 Implementation Notes.

1. The basic types which may appear are integer, character, boolean, real. There are also arrays of these of dimension less than or equal 2.

2. Nodes sI and sO are not explicitly mentioned for main graphs; in subgraphs they are called To-Node (sO) and From-Node (sI).

**Definition II.4. Exclusion Constraint**
An exclusion constraint, ec, on a set S, is a function on the power set P of S to {true, false} such that

i. ec({}) = true, ii. ec({s}) = true for every $s \in S$ and
iii. if U is a subset of P, ec(T) = true, and R is a subset of U, then ec(R) = true.

**Definition II.5.** Let ec be an exclusion constraint on S and let M be a subset of S. M is a **max-run set** if ec(M) = true and ec(N) = false for any superset N of M.

*Intuitively an exclusion constraint is a predicate which evaluates to true or false when acting on subsets of nodes of a UCG.*

**Lemma II.1.** Let ec be an exclusion constraint for S, then clearly

a. If T is any set for which ec(U) = true then there exists a max-run set containing U.
b. If {Mi} is the set of all max-run sets for the constraint ec, then $\cup$ Mi = S.
c. The set of max-run sets uniquely determines ec

**Definition II.6.** Let {Mi} be the set of all max-run sets for the constraint ec.

a. **Share**(ec) = $\cap$ Mi
b. If the Mi are pairwise disjoint then ec is said to be **mutex**. If they are singleton sets ec is said to be **strong mutex**
c. An exclusion constraint ec' for S' is **weaker than** exclusion constraint ec for S (**or ec is stronger than ec'**), if $S \supseteq S'$ and for every subset U of S', ec(U) = true implies ec'(U) = true. We write ec' < ec.

*Intuitively, the sets of nodes for which the exclusion constraint evaluates to true can execute in parallel with conformance to the specifications of the computation.*

**Definition II.7.** An **Exclusion Dependency**, e, has an associated set of nodes S(e), a type t = T(e), a name n = N(e), and an exclusion constraint ec = EC(e) on S(e). EC(G) is the set of exclusion constraints, ec, of the UCG, G

Two UCGs are **equivalent** if they differ at most in the names associated with their data and exclusion dependencies

*Intuitively, an exclusion dependency synchronizes the execution of a set of nodes to conform to the semantics of the computation as expressed in an exclusion constraint.*

## CODE 1 Implementation Notes

1. A node may take part in at most one exclusion dependency.
2. CODE 1.2 allows only two types of exclusion dependencies, those in which S(e)=Share(EC(e)) and those in which EC(e) is strong mutex.

## III. Semantics.

Roughly, one can picture the CODE 1.2 execution model as a dataflow graph which allows more flexible execution rules for some node types and which allows data sharing among the nodes. However, there are many other features which are further discussed below. The nodes are "black boxes" with their properties reflected only by their behavior at their interfaces. The semantics is in a sense only partly specified here. The internal state of SUCs and actual values appearing on arcs are not considered. Instead we restrict attention to defining the effect of firing a node on the enabling of other nodes and on termination.

### A. Data Dependencies

Data dependencies are like pipes, with a buffering capacity, that connect nodes. If there are items in the buffer then the dependency is said to be bound, otherwise it is unbound. The capacity of the buffers is an implementation issue that effects the semantics of the model. In CODE 1.2 the capacity is nominally infinite.

### B. Nodes

During execution, a node n of a UCG, G, may be in one of the states: idle, ready, or running. Legal transitions are from idle to ready to running to idle.

There are two types of node:

1. A **SUC**, s, is a unit of computation with the properties that
   a. When the computation changes state from running to not-running then data is placed on every output data dependency.
   b. When the computation changes state from ready to running one data item is removed from each member of In(s).

2. **A Switch** s a unit of computation with the properties that
   a. Data is consumed from one non-deterministically selected element of In(s) (which must be bound) when the state changes from ready-to-run to running.
   b. Data are placed in a subset of Out(s) when state changes from running to not-running

In CODE 1, switches are limited in the computations they may perform. They may test data on a single input arc only, and based on that data alone, they may distribute the data, unchanged to a subset of the output data dependencies. They may not participate in exclusion dependencies.

A node, n, is eligible to be promoted **from idle to ready** if sufficient data is available on its input dependency arcs. For SUCs this means that data is available on all input dependencies; that is, the state of d is bound for all d in In(s) {see below}. For switches it means that at least one element of In(d) is bound.

A node is eligible to be promoted from **ready to running** if doing so leaves the exclusion constraints in which it participates satisfied {see below}. On such a transition, one item is removed form each of the elements of In(n) for SUCs and from a single non-deterministically chosen element of In(n) for switches.

A node is always eligible to be promoted from **running to idle**. On such a transition, one item is placed in each element of Out(n) for SUCs and in some data-dependent subset of Out(n) in the case of switches.

### C. Exclusion Dependencies

If e is an exclusion dependency then there is assumed to be some object of type t which is shared by the elements of S(e) according to the discipline described by ec = EC(e). Consequently if V is a subset of S(e), ec(V) = true is the statement that it is permissible to have every node in V running, and every node of S - V not-running. This expresses a constraint on the ability of the computations associated with nodes in S(e) to access the shared data item whose type is t. Recalling the definition of exclusion constraint Definition II.3:

Condition i. means that the constraint is automatically satisfied if no node is running,
Condition ii means that the constraint is automatically satisfied if a single node is running,
Condition iii. means that if the constraint is satisfied then if a running node changes state to idle then the constraint is still satisfied.

An interesting issue is whether i, ii and iii. are reasonable. For example, iii. does not allow the case that a set of nodes may only run if some other node is running {as in a monitor construct}.

Exclusion dependency e1 is weaker than e2, if EC(e1) is weaker than EC(e2). This means that any set of nodes which e2 will allow to run together may run together under e1. We might say e1 is at least as permissive as e2.

### D. Subgraphs

Subgraphs are strictly a program-creation-time structuring device and have no other semantics. .

### E. State of a UCG

A **state,** s, of a CODE graph is defined by two functions

$f_s$: S(G) -> {idle, ready, running}
$g_s$: D(G) -> {bound, unbound} X N

The **initial state** is

F(n) = idle for all n in S(G) - sI
F(sI) = ready
G(d) = (unbound, 0) for all d in D(G)

The **terminated state** has F(n) = idle for all n.

### F. Firing rules {UCG state transitions}

In execution it is the responsibility of the runtime system to maintain the truth of all exclusion constraints. The way this has been done in CODE 1.2 is to use the following technique for starting computations:

> From the nodes that are ready, a single node is chosen and started. This node must have the property that if it is started, then no exclusion dependency will be invalidated. This is then repeated until no ready nodes exist.

## IV. Composition and Decomposition of Dependencies

To reach our goal of describing the composition of UCGs we must first describe how to compose and decompose dependencies. Ultimately these dependencies couple the interfaces of the components into parallel computation structures. We picture the components as being removed from larger configurations by cutting wires (decomposing dependencies) and as being recombined by splicing these wires together

**Definition IV.1. Compose two data dependencies**

If G is a graph with data dependencies d1 and d2 with d1 = ((s1, sO), n1:t1) and d2 = ((sI, s2), n2:t2), s1 <> sI and s2 <> sO, and if t = t1 = t2 then the graph G' obtained by **composing** d1 and d2 is created by deleting d1 and d2 from G and adding the dependency d = ((s1, s2), n':t).

**Definition IV.2. Decompose a data dependency**

If G is a graph with data dependency d and d = ((s1,s2), n:t) where s1 <> sI and s2 <> sO, then the graph G' obtained by decomposing d is created by removing the dependency d from G and adding the pair of dependencies d1 = ((s1, sO), n:t) and d2 = ((sI, s2), n:t).

*The intuition is that if an output data dependency of one node and an input dependency of another node match in type then the two dependencies can be joined to establish a data dependency between the node pair.*

**Definition IV.3. Projection of an exclusion dependency.**

Let e be an exclusion dependency. Let S' be any subset of S(e). The projection of e on S', written e proj S', is an exclusion dependency e' defined as follows:

1. S(e') = S'
2. ec' = EC(e') is defined as follows:
   If U is a subset of S(e') then ec'(U) = true iff ec(U) = true.
   ec' is also referred to as a projection of EC(e).
3. T(e') = T(e)
4. N(e') = N(e)

*A projection of an exclusion dependency maintains for subsets of nodes from the exclusion dependency the synchronization conditions for those subsets in the original exclusion constraint.*

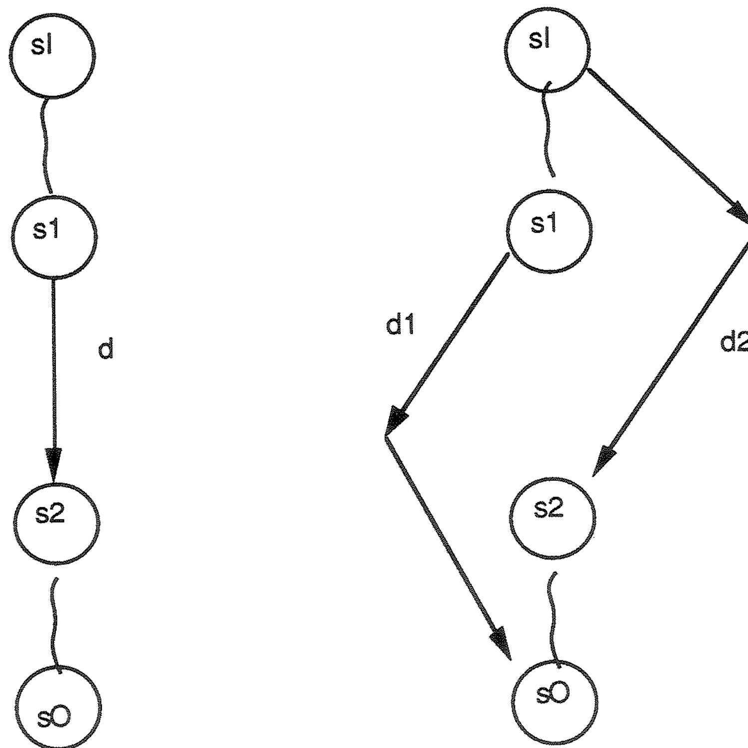



Figure 3
d is the composition of d1 and d2. d1 and d2 are the decomposition of d.

**Definition IV.4. Consistent exclusion dependencies**

If e1 and e2 are exclusion dependencies then they are consistent if

$$e1 \text{ proj } (S(e1) \cap S(e2)) = e2 \text{ proj } (S(e1) \cap S(e2))$$

Note that if $S(e1) \cap S(e2) = \{\}$, then e1 and e2 are consistent.

*Two exclusion dependencies can be composed into a single exclusion dependency if the projections of each with their intersection are equal.*

**Definition IV.5. Composing two consistent exclusion dependencies**

If G is a UCG and e1 and e2 are two consistent exclusion dependencies in G where t = T(e1) = T(e2) and if ec is an exclusion constraint such that EC(e1) = ec proj S(e1), and EC(e2) = ec proj S(e2) then the graph G' is obtained by **composing e1 and e2** as follows: e1 and e2 are removed from G and the exclusion dependency e is added where T(e) =t, N(e) = n', $S(e)=S(e1) \cup S(e2)$, and EC(e) = ec. If the max-run sets of e are exactly the max-run sets of e1 together with those of e2 then we say e is the **strongest** composition of e1 and e2, denoted $e_{st}$. If the max-run sets of e are all possible pairwise unions of a max-run set of e1 together with a max-run set of e2 then e is the **weakest** composition of e1 and e2, denoted $e_{wk}$. We say e is the composition of e1 and e2.

The following lemma assures us that the projections of a composition of two consistent exclusion dependencies are the original dependencies. Also, if e is any composition of e1 and e2 then e will be weaker than the strongest composition and stronger than the weakest composition of e1 and e2.

**Lemma IV.1** If e is a composition of e1 and e2 then

i. e1 = e proj S(e1)
ii. e2 = e proj S(e2)
iii. $e_{wk} < e < e_{st}$

*The exclusion constraint ec associated with an exclusion dependency e obtained by composition of two exclusion constraints e1 and e2 is is the range of constraints defined by the relations EC(e1) = ec proj S(e1) and EC(e2) = ec proj S(e2)*

### CODE 1.2 Implementation Note

In the case of share constraints Code 1.2 assigns ec to be $e_{wk}$. In the case of strong mutex constraints Code 1.2 assigns ec to be $e_{st}$.

## V. Subgraphs

**Definition V.1.** A **subgraph,** G', of a UCG, G, is a UCG such that

1. Nodes. $S(G) \supseteq S(G')$
2. Data Dependencies
   a. $d \in IND(G') \Rightarrow S(G') \supseteq S(d) \wedge d \in IND(G)$
      {An internal data dependency of G' is an internal data dependency of G with both its nodes in G'}
   b. $d' \in IOD(G') \Rightarrow$
      i. $d' \in IOD(G)$ or
      ii. there is a $d \in IND(G)$ and some d'' such that $d = d' \mid d''$
      {Input and output dependencies come from those of G or they are created by decomposing dependencies of G}
3. Exclusion Dependencies
   $e' \in E' \Rightarrow$ there exists $e \in E$ such that e' = e proj S'
   {Every exclusion dependency of G' is derived from one of G}

**Definition V.2.** A **full subgraph,** G', of a UCG, G, is a subgraph such that each of the '=>'s in 2 and 3 is replaced by '<=>'.

*The intuition is that a subgraph is a subset of nodes and a set of data dependencies which are either from the original graph or arise from decomposing data dependencies whose sinks and sources are*

*not both in the subset.. The subgraph is full if all possible dependencies are included. Exclusion dependencies of the subgraph must be projections of those in the original graph. Note that we allow single node exclusion dependencies. This is so they may be rejoined on a composition.*

### Code 1.2 Implementation Notes

1. The CODE/ROPE system allows any full subgraph of a UCG to be identified and stored in the library of reusable modules.
2. The CODE/ROPE system allows any full subgraph of a UCG to be identified and replaced with a single symbol. This allows hierarchical structuring of the graph under development.

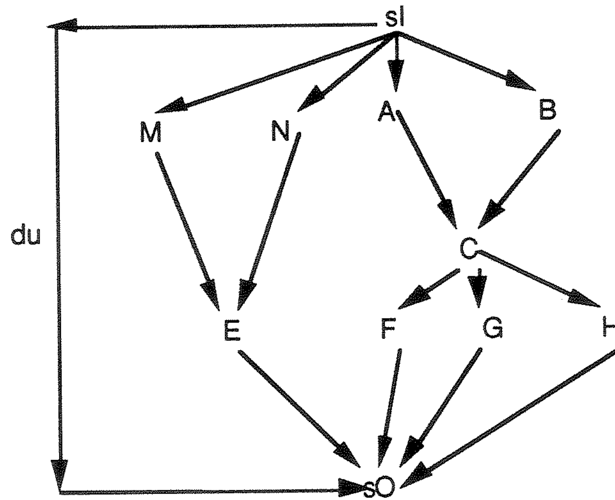



Figure 4. Full components of a UCG

a. {sI, A, B, C, F, G, H, sO}
b. {sI, A, B, C, sO}
c. {sI, C, F, G, H, sO}
d. Any single element with sI and sO
e. {s1, M, N, E, sO}
f. Any union of these with the property that they pairwise have no dependencies in common

**Definition V.3.** If G is a ucg and G' is a full subgraph then let the **complement**, H, of G' in G be defined as follows:

1. S(H) = (S(G)-S(G')) ∪ {sO, sI}
2. D(H) = {d | d ∈ D(G) and S(d) is disjoint from S(G')-{sO, sI}} ∪
   {d" | there is a d ∈ D(G) and d' ∈ D(G') such that d = d' | d"} ∪ {du}
3. E(H) = {e' | e' = e proj S(H) for some e in E(G)}.

{The complement is what is left if you remove the subgraph from the ucg; decomposing the dependencies which cross the boundary between S(G') and S(H').}

**Lemma V.1.** If G is a ucg and G' is a full subgraph then the complement, H, of G' in G is a full subgraph.

**Proof.**

1. H is a ucg

If s is a node then there was a path from sI to s in D(G), say (sO, d1, d2 , ... , dk, s). Let dj be the highest numbered dependency that is not in D(H). Then dj was decomposed naturally into d' and d" with Source(d")=sI and Sink(d")=Sink(dj). Hence, (d",....,dk,s) is a path from sI to s.
Other parts of the proof are similar.

2. H is a subgraph

This is long but mostly direct from the definitions.

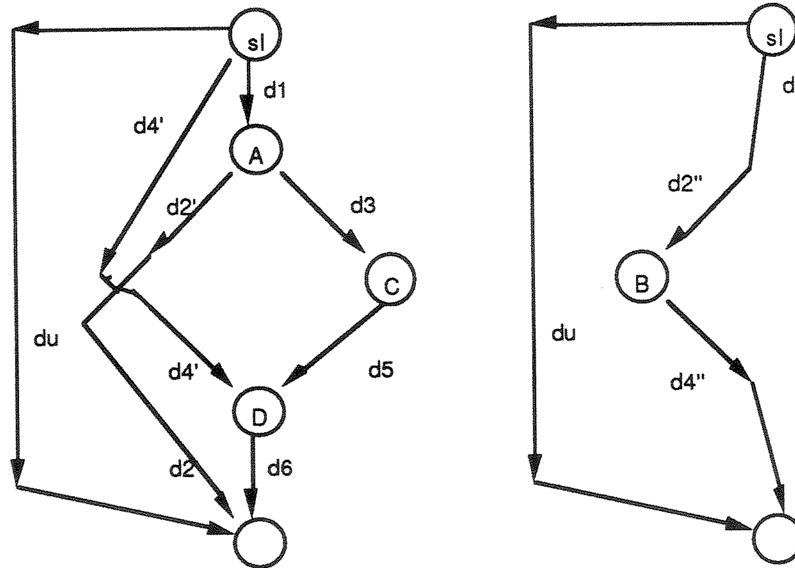



Figure 3
The Typical UCG of Figure 1 divided into complementary components

**Note.** The unique full subgraph of G based on (S(G)-S(G')) ∪ {sI, sO} is the complement of G' as defined above.

**Definition V.4.** G' is a **closed subgraph** of G if the following hold:

1. If $d \in$ IND(G') ∪ ((OD(G) ∩ OD(G')) , then D(G') ⊇ $Out_G$(Source(d))
   {if one child of $Source_G$ (d), is in G' then all are}
2. If $d \in$ IND(G') ∪ ((ID(G) ∩ ID(G')), then D(G') ⊇ $In_G$ (Sink(d))
   {if one parent of $Sink_G$ (d), is in G' then all are}
3. If $s \in$ S(G'), then E(G') ⊇ $E_G$ (s)

*Thinking of a ucg as a graph, a subgraph is closed if,*

*1. Given a node (other than sI) which is in the subgraph, if one of its children (in G) is in the subgraph G' then all children (in G) are included in the subgraph (closed under children)*

*2. Given a node (other than sO) which is in the subgraph, if one of its parents (in G) is in the subgraph G' then all parents (in G) are included in the subgraph (closed under parents)*

*3. Any exclusion dependency of G is completely contained in or completely outside the closed subgraph G'.*

In particular, note that ({sI, m, sO}, {d2, d4, du}, {}, T) is not closed in either Figure 4.i {the closed under children property is not satisfied} or Figure 4.ii is a subucg{the closed under parents property is not satisfied}. Node m in Figure 4.i is said to do **intermediate output**. In Figure 4.ii, m does **intermediate input**.

**Proposition V.1.**

If G' is a subgraph of G then G' is closed if $\forall s \in S(G')$

1. $D(G) \supseteq In_{G'}(s)$ or $ID(G') \supseteq In_{G'}(s)$
2. $D(G) \supseteq Out_{G'}(s)$ or $OD(G') \supseteq Out_{G'}(s)$
3. $D(G) \supseteq Ex_{G'}(s)$

**Discussion.** The full, closed subgraphs are natural objects for reuse. At a node s, either every internal input dependency of s is included in the subgraph or all have been replaced by connections to sI (similarly for output dependencies from s). That is, there are no data dependencies which cross the boundary of the subgraph except those coming from sI or sO. In addition, no data is shared across this boundary using exclusion dependencies.

Restriction of reuseable components to be only full, closed subgraphs would be very restrictive in practice. The composition operators defined and described following allow instantiation of components with exclusion dependencies composed across subgraph boundaries.

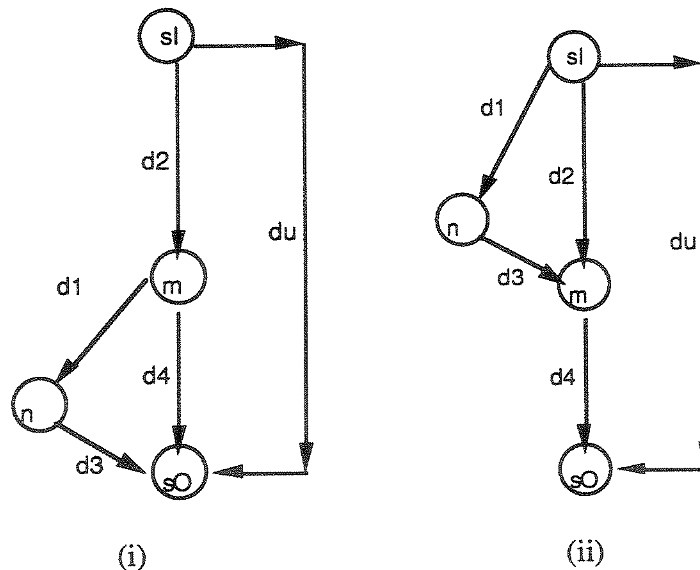



Figure 4

**Definition V.5.** A **subucg** G' of G is a ucg such that

1. $S(G) \supseteq S(G')$
2. $D(G) \supseteq D(G')$
3. $E(G) \supseteq E(G')$

**Discussion.** A subucg is literally a "subset" of G. It might or might not be a subgraph. That is to say, unlike a subgraph, there is no requirement that dependencies of G with one node not in the subucg be acknowledged by inserting new I/O dependencies in the subucg.

**Definition V.5.** A **normal subucg** G' of G is a subucg which is also a full, closed subgraph.

**Proposition V.2.**
Let G be a ucg and let G' and G" be normal subucgs, then

1. G' $\cap$ G" is a normal subucg
2. G' $\cup$ G" is a normal subucg
3. The complement of G' is a normal subucg

**Definition V.6.** A normal subucg is **minimal** if it contains no proper non-trivial normal subucg.

**Lemma V.2.**
Two minimal normal subucgs are equal or have trivial intersection.

**Proposition V.3.**
The union of all minimal normal subucgs of G equals G
**Proof.**
For any node, dependency, or exclusion dependency, x, there is some normal subucg containing it, namely G itself. Take the intersection, G', of all such subucgs. If G' is minimal, we are through. Suppose that G" is minimal normal subucg properly contained in G' and not containing x. Let H be the complement of G'. Then x $\in$ H, and H $\cap$ G' is a normal subucg; contradicting the definition of G'.

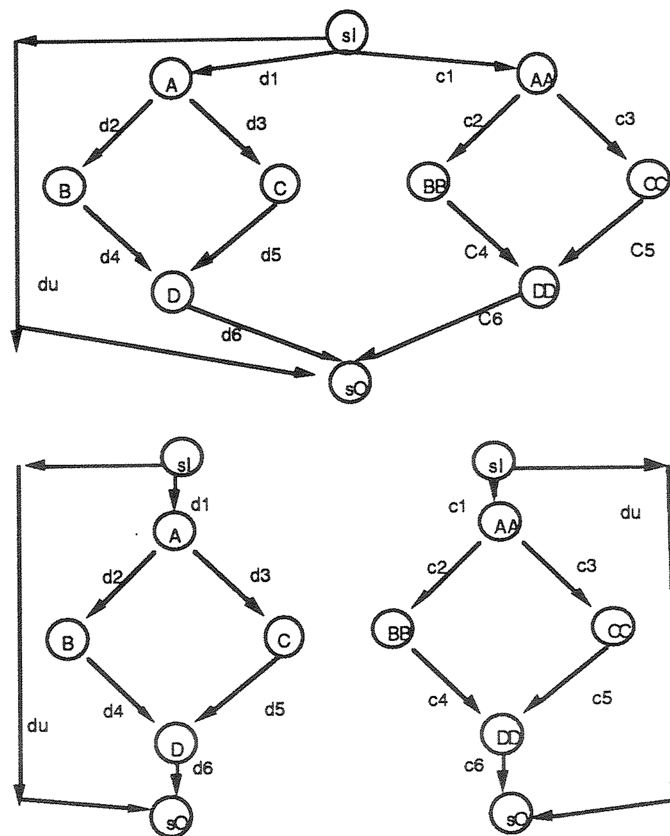



Figure 6
A UCG written as the union of its minimal normal subucgs

**Definition V.7.** The **degree of independent parallelism** of a UCG, G, denoted ip(G), is the cardinality of the set of minimal normal subucgs of G.

*The degree of independent parallelism corresponds to the intuitive idea of decomposing the ucg into the largest possible collection of independent subgraphs (presumably so they could be run on different machines)*

## VI. Composition of UCGs

Composition is an attempt to capture formally the idea of taking two ucgs and connecting some of their I/O dependencies and exclusion dependencies and thereby creating a new ucg. This process is the essential step in building software from components. The connection of data dependencies is specified by a pair of functions. The connection of exclusion constraints is specified by giving the two exclusion dependencies to be combined and a description of how to combine their constraints. An informal description of the calculus of composition for CODE/ROPE has been given previously [BRO89].

**Definition VI.1.** Let A and B be sets. We say that g is a **partial 1-1 function** from A to B (denoted g : A -> B) if A $\supseteq$ dom(g), B $\supseteq$ ran(g), and g is 1-1 from dom(g) to ran(g). If dom(g) = { } (and hence ran(g)={ }), then g is the **null** function.

**Definition VI.2.** Let G and H be two UCGs such that

i. $S(G) \cap S(H) = \{sI, sO\}$

ii. $D(G) \cap D(H) = \{du\}$

iii. $E(G) \cap E(H) = \{\}$

and let Gio, Goi, Ge be partial 1-1 functions (called **connecting functions**) such that

Gio : ID(G) -> OD(H) and such that T(d) = T(Gio(d))

;Gio describes which inputs of G to connect to which outputs of H

Goi: OD(G) -> IG(H) and such that T(d) = T(Gio(d))

;Goi describes which inputs of H to connect to which inputs of G

Ge: E(G) -> E(H) and such that T(e) = T(Gio(e))

;Ge describes which exclusion dependencies of G and H should be joined.

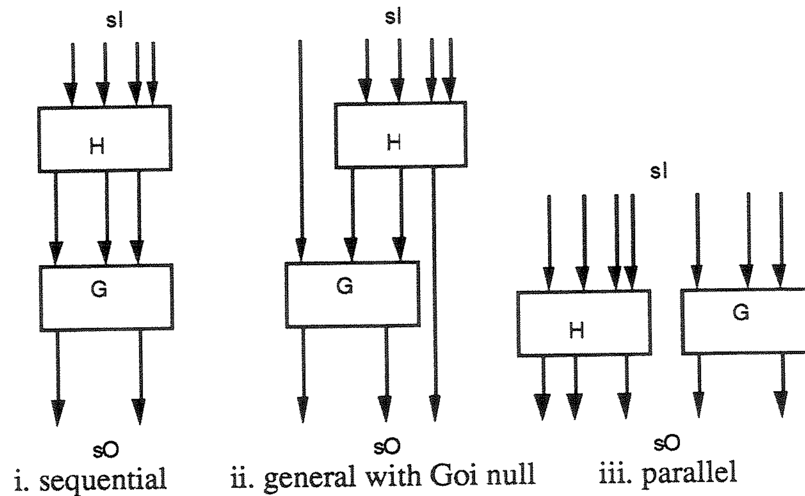
In addition, let E be a set of exclusion constraints, one such constraint ec for each pair (e1, e2) with e2 = Ge(e1). This exclusion constraint ec is defined on $S(e1) \cup S(e2)$ such that EC(e1) = ec proj S(e1) and EC(e2) = ec proj S(e2).

Then the **composition** of G and H with respect to (Gio, Goi, Ge, E), $G\ \Delta_{(Gio,Goi,Ge,\ E)}\ H$, is a UCG K defined as follows:

1. $S(K) = S(G) \cup S(H)$
2. $IND(K) = \{du\} \cup IND(G) \cup IND(H) \cup$
   $\{d \mid d \text{ is the composition of } d' \text{ and } Gio(d'), \forall d' \in dom(Gio)\} \cup$
   $\{d \mid d \text{ is the composition of } d' \text{ and } Goi(d'), \forall d' \in dom(Goi)\}$
3. $ID(K) = (ID(G) \cup ID(H)) - (dom(Gio) \cup ran(Goi))$
4. $OD(K) = (OD(G) \cup OD(H)) - (dom(Goi) \cup ran(Gio))$
5. $E(K) = ((E(G) \cup E(H)) - (dom(Ge) \cup ran(Ge)) \cup$
   $\{e \mid e$ is the composition of e' and Ge(e') with respect to ec, $\forall e' \in dom(Ge)$, where ec is the associated exclusion constraint in E$\}$

**Definition VI.3.** The composition is called

i. **sequential** (Fig. 5.i),if Gio is total and onto, and Goi is null
ii. **parallel** (Fig 5. iii) if Gio and Goi are null
iii. **independent** if Ge is null.
iv. We say that K is constructed by **embedding G in H** (Fig. 7) if Gio and Goi are total (every input of G is connected to an output of H and every output of G is connected to an input of H)




i. sequential ii. general with Goi null iii. parallel

Figure 5

Figure 5.ii illustrates a general composition with Goi = null and Figure 6 is a general composition with Gio and Goi both non-null.

**Lemma:VI.1** Let K be the composition of G and H with respect to (Gio, Goi, Ge, E). Then

i. G and H are full subgraphs of K.
ii. G and H are normal if and only if the composition is independent and parallel

## CODE 1.2 Implementation Note

Code 1.2 does not incorporate either the ability to intermix graphs specified in the calculus of composition and the graphical representation or an automation of composition operators when subgraphs are instantiated. Both capabilities will be incorporated in CODE/ROPE 2.

## VII. Conclusion

The development of CODE and ROPE has been focused and directed by interaction with the formal model of parallel computation upon which they are founded. The role of CODE and ROPE has been to identify the requirements for a practical parallel programming environment while the formal model has first enabled a consistent evolution of the programming environment and then supplied directions for extending the programming environment to include significant capabilities not in view when the system was first being developed.

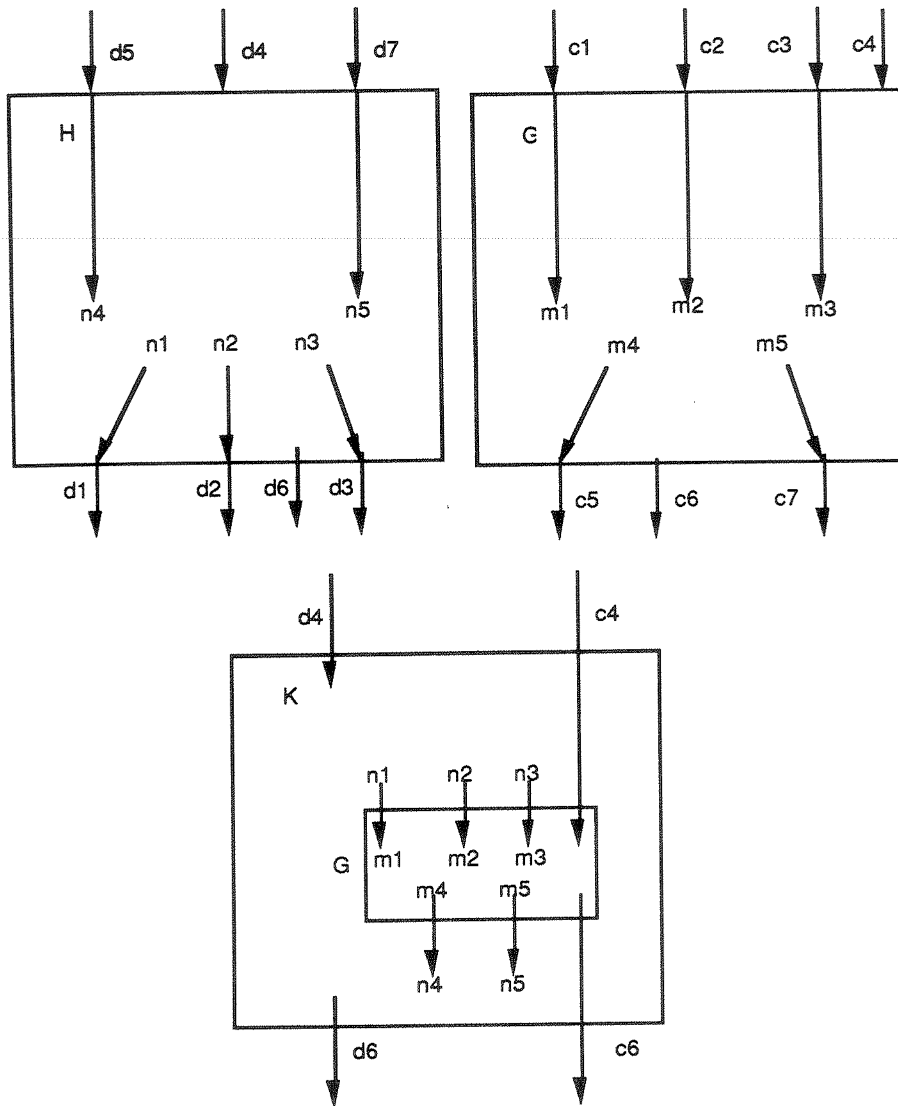



Figure 6. A general case of composition of G and H

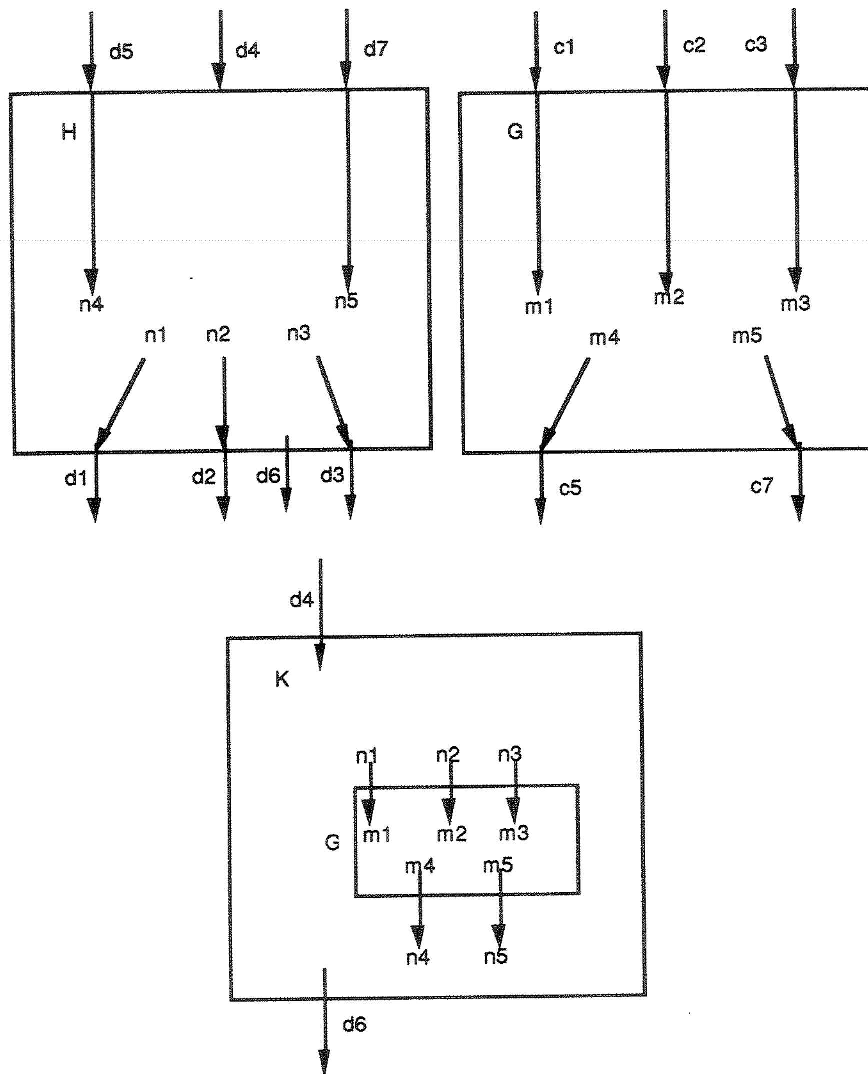



Figure 7. G embedded in H

**References**

[AZAM88] Azam, M., and Lin, C., "Programming with CODE: A Computation Oriented Display Environment", Department of Computer Sciences, The University of Texas at Austin, Oct. 1988.

[BATO87] Batory, D., "A Molecular Database Systems Technology", TR-87-23, Department of Computer Sciences, The University of Texas at Austin, June 1987.

[BATO88] Batory, D., Barnett, J., Garza, J., Smith, P., Tsukuda, K., Twichell, B., and Wise, T., "GENESIS: An Extensible Database Management System", IEEE Transactions on Software Engineering, Nov. 1988.

[BROC81] Brock, J., and Ackerman, W., "Scenarios: A Model of Non-determinate Computation", Lecture Notes in Computer Science #107, Springer-Verlag, New York, 1981.

[BROW84] Browne, J., "Comparison of Process/Transaction and Object-Oriented System Structures", Unpublished Manuscript, Dec. 1984.

[BROW85] Browne, J., "Formulation and Programming of Parallel Computations: A Unified Approach", In Proceedings of IEEE International Conference of Parallel Programming, 1985.

[BROW86] Browne, J., "Framework for Formulation and Analysis of Parallel Computation Structures", Parallel Computing 3, 1986, 1-9.

[CHA89] Chandy, M. and Taylor, S., "PCN", Technical Report, California Institute of Technology, 1989.

[DAVI82] Davis, A., and Keller, R., "Data Flow Program Graphs", IEEE Computer, Feb. 1982.

[DIJK69] Dijkstra, E., "Structured Programming", Classics in Software Engineering, edited by E. Yourdon, p43-48, Yourdon Press, New York, 1979.

[GLIN86] Glinert, E., "Towards 'Second Generation' Interactive, Graphical Programming Environments", 1986 IEEE Computer Society Workshop on Visual Languages, June 1986.

[GOGU86] Goguen, J., "Reusing and Interconnecting Software Components", IEEE Computer, Feb. 1986.

[IEEE85] IEEE Computer August 1985 Issue.

[IEEE86] 1986 IEEE Computer Society Workshop on Visual Languages, Dallas, Texas, June 1986.

[KARP66] Karp, R., and Miller, R., "Properties of a Model for Parallel Computations: Determinancy, Terminations, Queueing", SIAM Journal of Applied Math., Vol. 14, No. 6, Nov. 1966.

[LEE88] Lee, T., and Lin, C., "ROPE User's Manual: A Reusability-Oriented Parallel-programming Environment", Department of Computer Sciences, The University of Texas at Austin, Oct. 1988.

[PARN72] Parnas, D., "A Technique for Software Module Specification with Examples", Communications of the ACM, May 1972.

[PONG86] Pong, M., "A Graphical Language for Concurrent Programming", 1986 IEEE Computer Society Workshop on Visual Languages, June 1986.

[PRI82] Prieto-Diaz, R., Neighbors, J., "Module Interconnection Languages: A Survey", U. Calif. Irvine, ICS Technical Report 189, Aug. 1982.

[REIS84] Reiss, S., "Graphical Program Development with PECAN Program Development Systems", Proc. ACM Sigsoft-Sig
plan Software Engineering Symp. Practical Software Development Environments, Apr. 1984.

[SOBE88] Sobek, S., Azam, M., and Browne, J., "Architectural and Language Independent Parallel Programming: A Feasibility Demonstration", In Proceedings of IEEE International Conference of Parallel Programming, 1988.

[VAN86] Van Sickle, L., "PIPER - A Knowledge Based System for Program Reusability", Dissertation Proposal, Computer Science Department, University of Texas at Austin, Aug. 1986.

## CODE project publications

[AZA88] M. Azam, C. Lin, "Programming with CODE: A Computation Oriented Display Environment", Department of Computer Sciences, The University of Texas at Austin, Oct. 1988.

[BRO89a] J.C. Browne, M. Azam, S. Sobek, "CODE: A Unified Approach to Parallel Programming", IEEE Software, July 1989.

[BRO89b] J.C. Browne, J. Werth, and T.J. Lee, "Intersection of Parallel Structuring and Reuse of Software Components:A Calculus of Composition of Components for Parallel Programs", International Conference on Parallel Processing, 1989.

[BRO89c] J.C. Browne and J. Werth, "Software Engineering of Parallel Programs in the Computation-Oriented Display Environment", 1989 Minnowbrook Conf. on Software Engineering of Parallel Programs.

[BRO89d] J.C. Browne and J. Werth, "Software Engineering of Large Grain Parallel Programs", 1989 Workshop on Large Grain Parallelism, Carnegie-Mellon.

[BRO90] J. C. Browne, T.J. Lee and J. Werth, "Experimental Evaluation of a Reusability Oriented Parallel Programming Environment," IEEE Transactions on Software Engineering, Vol 16, No. 2, 1990.

[LEE90] T.J. Lee, "Software Reuse in Parallel Programming Environments", Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, 1989.

[NEW91] P. Newton, " Translation from a Declarative Model Of Parallel Computation to Multiple Procedural Models", Dissertation in progress, Dept. of Computer Sciences, University of Texas at Austin, 1990.

[SOB88] S. Sobek, M. Azam, and J.C. Browne, "Architecture and Language Independent Parallel Programming: A Feasibility Demonstration", Proc. of IEEE International Conference of Parallel Programming, 1988.

[SOB90] S. Sobek, "A Constructive Unified Model of Parallel Computation", Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, 1990.

[WER90] J. Werth, D. Banerjee, J.C. Browne, R. Jain, S. Lin, P. Newton, R. Rao, S. Sobek, CODE 1.2 User Manual and Tutorials, TR-90-35, Department of Computer Sciences, University of Texas at Austin, November 1990.