Anonymous Class in Declarative Mathematical Modeling

Benjamin A. Allan and Arthur W. Westerberg
Department of Chemical Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA

Abstract

Object-oriented, equation-based process modeling systems [2, 4, 5, 9] can be very helpful in producing small novel mathematical models, i.e. involving a few tens to few hundreds of equations. These systems have so far not been shown to improve the speed of creating large, novel models involving several tens of thousands of equations, the kind that must be based on the application and modification of libraries. Compile times and memory usage become bottlenecks when a researcher or process designer is frequently recompiling a large model or when a synthesis program is constructing many hundreds to thousands of alternative process models.

We report automatic compilation algorithms which reduce times from minutes to seconds by taking into account the rich hierarchical semantics of user-written object definitions. We report algorithms that reduce overall computer simulation memory requirements by more than half and also make compilation of equations all the way to loop-free, binary form practical. Our discovery of the anonymous class of partially compiled objects allows these reductions in CPU time and core memory usage. Preliminary results indicate that interactive, interpreted modeling systems may soon be as fast as less flexible batch systems which rely on very large libraries of precompiled binary modules. The ideas we present are experimentally verified here and elsewhere [3] to allow interactive manipulation of 100,000 or more equations. The experiments have been conducted in the ASCEND IV[1] system [1, 2, 3].

Introduction: Discovering similar objects to improve performance and translation

Virtually all large problems are *or ought to be* constructed by repeating small structures which are

1. ASCEND IV is free, documented software available for Windows and UNIX systems via http://www.cs.cmu.edu/~ascend.

better understood and adding or replacing a few features. A distillation column is computed by iterating over a set of trays. A flowsheet is computed by iterating over a set of columns and other units. A production schedule is computed by iterating over a set of plants and time periods. We can discover these repeated structures automatically, even though they are usually anonymous, and then use that knowledge of structure to reduce memory and CPU requirements for large models.

We introduce a general method to regain the speed and compactness of machine code while retaining the flexibility of the interactive systems. ASCEND III [10] and similar systems, sacrifice the extreme speed and compactness of compiled machine code to obtain a flexible representation which supports interactive model respecification and the interactive solution of any arbitrary subproblem. The general method we propose requires determining the *anonymous class* of each compiled object. Models written in reusable chemical engineering software libraries generally leave constants which determine the final size of the compiled model instance unspecified, such as the number and identity of chemical species in a stream or the number of stages in a distillation column. Two tray objects compiled from the same formal class definition are of distinct *anonymous class* unless the constants specified to each tray at compilation time are identical.

Background and range of application

Before explaining the our new methods, we must introduce some simple concepts well-known in computer science.

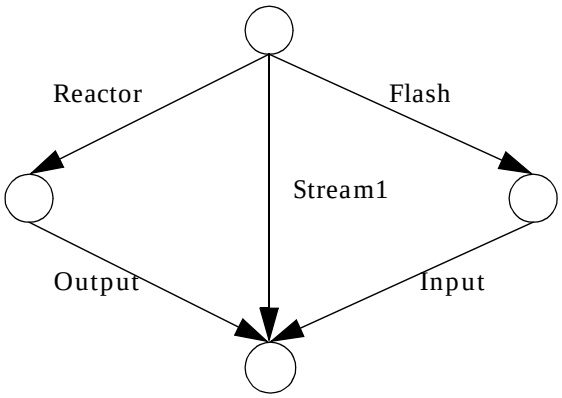
Graphs of shared data structures

Hierarchical, object-oriented modeling systems create data structures that are representable as directed acyclic graphs (DAGs). Higher level models in a flowsheet such as unit operations refer to and often share lower level models such as stream or physical property calculations. We illustrate this with the code and graph shown in Figure 1. The names specified for parts are shown on the links of Figure 1. The stream exiting the reactor and entering the flash is constructed and passed to both units in an object-oriented paradigm. The stream or either of the two units can be solved alone or as part of the flowsheet since each object in the graph can be isolated with all its subgraph. The same DAG structure is obtainable using the data structures of any language which

has implicit or explicit passing of pointers, such as FORTRAN, C++, or Java. We shall see that

```
MODEL FlowSheet;
      Stream1 IS_A stream;
      Reactor IS_A partial_reactor(Stream1);
      Flash IS_A partial_flash(Stream1);
END FlowSheet;
```

(a)



(b)

Figure 1 Directed acyclic graph of two units and their connecting stream.

this DAG structure, ubiquitous in object-oriented process modeling systems, makes determination of the exact subclass of constructed objects challenging.

Formal class

Each compiled object in a declarative modeling language is an instance constructed from a formal class definition. The statements of the formal class name each part and define the superclass from which each part will be constructed, as seen in Figure 1a. Most such languages [6, 7, 8] allow the definition of sets and arrays of parts indexed on these sets. Each array is counted as a single part. The final value of a set, and hence the array sizes, may not be included in the formal class. This defers the array size definition until the final application of the formal class as part of a larger problem; most software of substantial complexity achieve reusability through this deferred array range definition. In languages which support deferred binding, such as ASCEND, additional statements, possibly in a higher scope, also may at compilation time change a part of the compiled object from that part's initial superclass into any of its available subclasses. Deferred set definition

and part subclass specification change the class of the set or part but do not change the formal class of the enclosing compiled object.

Anonymous class detection

The formal class of an object does not completely determine its internal, hierarchical structure, as just noted. The formal class of an object specifies only the name and formal class of each of its parts (children). If the modeling language supports any form of object sharing either directly or indirectly, then we must include knowledge of the deep structure (which internal parts are shared and how they are interconnected) in determining the anonymous class of each object. In a user-directed approach to exploiting structural similarities [1], the user must understand anonymous class subtleties and must identify to the language compiler the important repeated structures. In an automatic approach, we must detect the anonymous class of each object in the instance hierarchy in order to identify the repeated structures.

The results of our first experiment to test the potential impact of the user-directed approach and an automatic approach are given in Table 1. In this experiment, we use an optimistic algorithm for determining the anonymous class of each object. The optimistic algorithm checks the formal class of the object being classified and the already determined anonymous class of each part in the object to determine the object's anonymous class. Once the anonymous classes are determined, we compile and share equation structures as already indicated.

Table 1: Optimistic equation compilation time and memory consumption

17,500 equation C3 splitters (194 trays)	No equation sharing	User directed equation sharing	Anonymous class guided equation sharing
CPU seconds	83.1	5.4	3.6
% CPU reduction	0	94	96
Megabytes	26	14	11
% Memory reduction	0	46	58

By finding the minimum set of unique equations we achieve memory efficiency of the same order

as hand-coded FORTRAN[1]. Since the set of unique equations is small, it becomes reasonable to consider compiling them all the way to loop-free machine code, thus reaching the maximum possible function and gradient evaluation speed. Compiling machine code separately for each of 100,000 equations will break most compilers outright, and can take hours on any common serial processor/compiler combination of which we are aware. Thus, equation sharing is not just desirable, but *required* if equation-based modeling language tools are to be a feasible alternative to large binary libraries of hand-coded unit operations. These results provide an upper bound on the cost reductions that anonymous classification knowledge can yield for equation construction.

Interpretation of equations and anonymous class

We assert that exploiting anonymous class information is easy, and that we can determine a reduced set of unique equations from it. We will show this to be true for the ASCEND language and compiler; we believe the necessary properties cited hold for nearly every equation-based system. These properties are:

1. an equation definition is a symbolic string of operators and operands, possibly indexed over sets, contained in a model class. All operands and sets refer to variables within the scope of the class, possibly variables found down in some member (part) of the class.
2. an equation instance is a local attribute (child) of exactly one model instance.
3. an equation instance includes a list of byte codes derived from the equation definition with all sets expanded. It is this set of byte codes we will avoid recreating by discovering anonymous classes.
4. Each byte code reference to a variable resolves to an address in a locally defined lookup table [1].
5. The lookup table is constructed from the variable names in the equation definition by resolving each name in the instance tree of the model instance.
6. The formal class of an instance determines the child index that corresponds to a child name in every instance of that class. So if a variable named “flow” in the class “Stream” has the child index 3 in one instance, then it will have that index in all instances of Stream. This property is also used in deriving our anonymous classification algorithms.

Consider building equations in a set of model instances which have identical anonymous classes (identical DAGs and sets). By knowing that an equation instance is the child of exactly one model instance, we know that all parent instances of that equation instance have the same anonymous

1. While the order is the same, the coefficient is somewhat higher to support the flexibility of arbitrary interactive problem specifications and subproblem management, features not commonly available from black-box software.

class; the interpretation of the equation definition cannot be ambiguous. By knowing that the DAGs and sets are identical and by property six, we know that two byte code lists derived from the same equation definition must be the same and can thus be shared. We can record the index path found as we resolve each variable name in the symbolic equation to a variable instance while building byte codes for that equation in the first model instance. We can simply trace the recorded index paths through the second and succeeding model instances to fill in the lookup tables with corresponding variable instances. This byte code sharing and path tracing eliminates reprocessing the symbolic form of the equation.

This set of equations is the smallest that can be determined from the anonymous class of a model instance; the size of this “unique” set determined by class is given for our examples as column Nuc of Table 5. There is a set of unique equations which is still smaller than can only be obtained by sorting all the equations throughout the DAG. For example many different formal classes may connect pairs of variables with an equation of the form X = Y where X and Y here are place holders for any pair of names defined in a class definition. Searching for these equation byte code isomorphs becomes feasible in the small set of “unique” equations determined by the anonymous class mechanism. This sort would not be practical if every equation in a large DAG had to be sorted. Column Nus of Table 5 gives the number of equations with distinct byte code lists. It is this “byte code unique” set of equations only that should be translated to FORTRAN, symbolically checked for convexity, or otherwise symbolically manipulated.

Compiler instruction scheduling

Computer language compilers work in several passes, or phases, to make implementation of the compiler easier and more efficient and also to make diagnosis of incorrect input more humane. The semantics of an equation are determined entirely by the combination of the equation’s symbolic form and the structure of other objects in the same scope. Thus, compiling equations can be deferred until all compilation of the model and variable object structures is finished. This allows the detection of all model structure errors to be carried out before any CPU time is wasted on building the equations in an erroneous model. Similarly, compilation of each equation need be attempted only once. If compilation fails, then some variable simply does not exist and the user can be informed precisely what is wrong. This eliminates compiler retry of erroneous equations

and saves the user time while new model constructions are being debugged.

We make reductions in compilation time by separating equations into a second compiler pass as described in [3]. Far larger gains are available, however, if we can detect the repeated model structures and compile only one copy of the equation for each kind of structure (anonymous class) discovered. All exactly similar models can share this copy of the equations, as would occur in a painstakingly handcrafted FORTRAN library. Discovering the minimum set of equations needed to represent a large model can potentially reduce the communication volume when transmitting models or parts of models among networked processors. Relation sharing is recognized as significant in the process of user-directed model copying in [1]. A user-directed approach cannot, however, make maximum use of repeated structures and is vulnerable to user misdirection.

Algorithms for rigorous classification

Why did we describe the classification algorithm referred to in Table 1 as *optimistic*? We did so because models compiled with object passing, or with any other way of sharing an object pointer among several model parts, form DAGs and not trees. On a DAG it is not sufficient to check just the anonymous class of each part in an object when classifying the object. Figure 2 illustrates why this is the case with the simplest possible example.

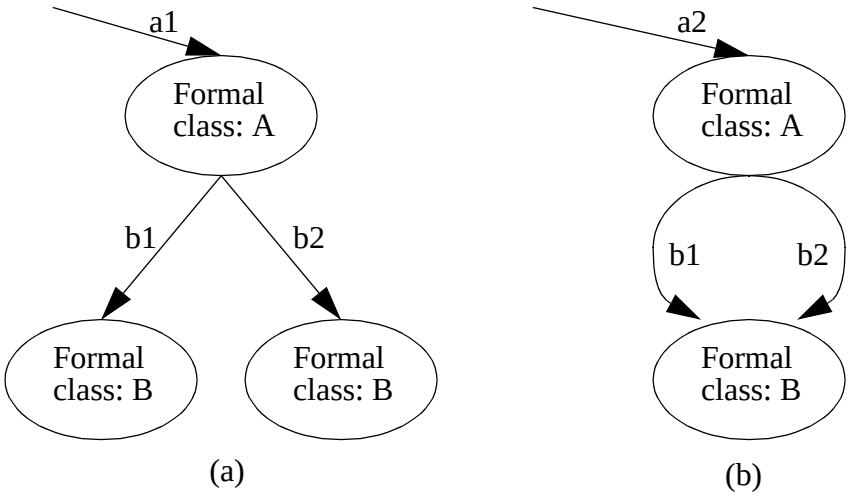


Figure 2 Counterexample instances to an optimistic classification algorithm

According to our optimistic algorithm, the anonymous class of objects a1 and a2 is the same. a1

and a2 are both of class A and both have two children of class B. However, a1 and a2 name clearly different data structures with different semantics. Consider an equation in the definition of class A, e.g. `b1 = b2`. This equation relates two variables, b1 and b2, in the object named a1, but merely states a tautology in the object named a2. Unfortunately, this *kind* of subtlety *does* occur in the flowsheeting models we have tested. A rigorous classification algorithm is required which takes deep structure sharing information into account.

A situation like that of Figure 2 (but obscured in layers of structure) is found in the thermodynamics library we use with ASCEND. Imagine a user trying to find a compiler's optimization error of this sort buried in a model of 100,000 equations and several hundred objects. Our optimistic strategy is not acceptable. Any mathematical programming system which passes pointers in the construction of its object data structures needs to account for the subtle effects of anonymous class within DAG structures in order to use the equation sharing required for scale-up to large models.

The detection of anonymous classes is practical, and the exploitation of anonymous classes is easy. We present results for two examples in Table 2; this data set suggests our approach is practical. We shall present more extensive test data to show that the algorithms are scalable in Table 4 after we describe the algorithms. We then examine the polynomial order of each algorithm

Table 2: Pass two equation compilation time and memory

SunOS/Sparc 5-110	No equation sharing	User directed equation sharing	Anonymous class guided equation sharing
17,500 equation C3 splitters (71 unique equations)			
CPU seconds	96.5	5.8	13.5
% CPU reduction	0	94	86
Megabytes (total simulation)	26	14	11
% Memory reduction	0	46	58
79,500 equation ethylene plant (177 unique equations)			
CPU seconds	923	34	59
%CPU reduction	0	96	94
Megabytes (total simulation)	192	60	60

Table 2: Pass two equation compilation time and memory

SunOS/Sparc 5-110	No equation sharing	User directed equation sharing	Anonymous class guided equation sharing
% Memory reduction	0	69	69

involved and show how they may be expected to behave in a linear or mildly quadratic fashion in practical applications.

We derive an anonymous classification algorithm by induction. Assume that we have correctly determined the anonymous class of all the children of an object, O, in a DAG. Algorithm 1.1 then yields the new anonymous class of O or puts O into the list of objects which are already known to share its anonymous class.

Algorithm 1: Sorting objects in a DAG by anonymous class

1 If the hierarchy forms a DAG
Apply Algorithm 2 to derive the minimal subgraph identity information needed to compare efficiently the deep structure between any two nodes of like formal type.
Else,
EXIT. (This algorithm is not needed to build formally typed trees efficiently.)
2 Initialize (to empty) GF2ACL, a hash table keyed by formal class name or a contiguous array indexed by some unique integer property of each formal class.
3 Set ACCount to 1.
4 Visit once each object node, O, in the DAG in a depth-first, bottom up order, applying algorithm 1.1 at each node to derive that node's anonymous class. The visitation must be bottom-up because Comparator 1 requires information about the anonymous classification of the child nodes of O.

Algorithm 1.1

1 Query the object, O, for its formal class, F.
2 Fetch from GF2ACL the list of anonymous classes, ACList, already derived for the formal class F.
3 If ACList does not exist:
Create ACList.
Create an anonymous class descriptor, ACDesc, with O as its exemplar, E, and ACCount as its index.
Increment ACCount.
Add O to the complete list of objects, LO, belonging to ACDesc.
Record ACDesc on node O as its anonymous class.
Add ACDesc to ACList.
Add ACList to GF2ACL.
Continue with the next node in the DAG.
Else,
Search ACList using Comparator 1 for an anonymous class descriptor, ACDesc, whose exemplar, E, matches O.
If such an ACDesc does not exist in ACList

Create ACDesc with O as its exemplar, E, and ACCount as its index.
Increment ACCount.
Record ACDesc on node O as its anonymous class.
Insert ACDesc in ACList in sorted order using Comparator 1.
Else,
Add O to the list of objects, LO, belonging to ACDesc.
Record ACDesc on O as its anonymous class.
4 Continue with the next node in the DAG.

Comparator 1 is the constructive definition of anonymous class for a mathematical modeling object, though it in principle applies to any form of software object, not just mathematical ones. Comparator 1 returns -1 (greater-than) or 1 (less-than) if two structured or scalar objects are different, and it returns 0 if the two objects are semantically equivalent (are of the same anonymous class and deep structure), differing only by the values of contained scalar variables. The scalar variable *values* may change in succeeding mathematical analyses, so they should not be considered part of any classification scheme, be it formal or anonymous. Comparator 1 is transitive, ((O1 < O2) AND (O2 < O3)) ==> (O1 < O3), so it can be used for sorting lists of anonymous class descriptors.

Comparator 1: Comparing two objects, Oa, Ob of the same formal class

1 #Comment: We define variables and equations as atomic, not having an anonymous class distinct from formal class.
If Oa is a scalar variable or relation, then
return 0.
2 #Comment: Constant booleans, integers, reals, complexes, character strings, and sets of integers or strings fall in this category. The latter, sets, are frequently used to determine the size of associative arrays in equation-based modeling languages. We assume any such language implementation can define the comparators greater-than (>) and less-than (<) for its scalar constant classes including sets.
If Oa is a scalar constant, then
If Oa.value < Ob.value return 1.
If Oa.value > Ob.value return -1.
return 0.
3 #Comment: We assume that associative arrays are indexed over sets containing discrete scalar values. We assume that an associative array may contain elements (children) of diverse formal classes. We define two associative array objects to be different if their index sets are different, or if, elementwise, the arrays contain objects of distinct anonymous class for any index I, or if their subgraph identities determined by Algorithm 2 are different. Contiguous (instead of associative) arrays of identical elements are a trivial case and can be compared by checking for identity of the array index range and similarity of the first element of the two arrays in question.
If Oa is an associative array, then
If Oa.index-set < Ob.index-set, return 1.
If Oa.index-set > Ob.index-set, return -1.

For each element I in Oa.index-set:
Get the anonymous class, ACDesc1, from Oa.I.
Get the anonymous class, ACDesc2, from Ob.I.
#Comment: The following two comparisons are arbitrary but consistent.
If ACDesc1.index < ACDesc2.index, return 1.
If ACDesc1.index > ACDesc2.index, return -1.
Next I.
return Comparator-2(Oa.Subgraph-Identities, Ob.Subgraph-Identities).

4 #Comment: We assume that the formal class of a container object determines the name and superclass of each of its contained parts (children in the DAG). We assume that an integer indexing of the children exists and that this indexing is the same for all instances of the container so that child J of Oa corresponds to child J of Ob. A child J of an object O may be of a formal class which is a subclass of the superclass specified in the formal definition of O.

Any other object must be a container object (an instance of a model class), so
For each child J of Oa:
Get the anonymous class, ACDesc1, from child J of Oa.
Get the anonymous class, ACDesc2, from child J of Ob.
#Comment: The following index comparisons are arbitrary but consistent.
If ACDesc1.index < ACDesc2.index, return 1.
If ACDesc1.index > ACDesc2.index, return -1.
Next J.
return Comparator-2(Oa.Subgraph-Identities, Ob.Subgraph-Identities).

5 Exit with error. O is not a valid input to this comparator.

We must explain the simple concept of subgraph identity within a DAG of formally classed objects so we can explain Comparator 2. Our goal for Algorithm 2 is to reduce the pattern of deep interconnections within a DAG to a compact, *strictly local* form that is easily compared. We gave a trivial example containing a subgraph identity in the object named a2 in Figure 2. We show how this identity can be represented as a list of names in Figure 3. Here the anonymous class of the

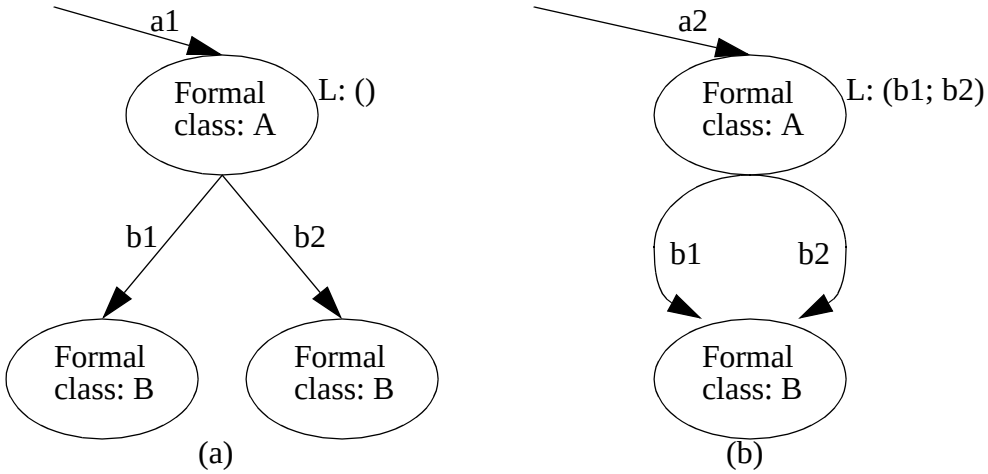
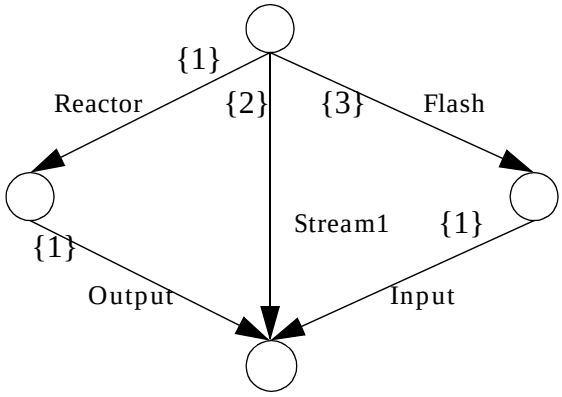


Figure 3 Subgraph identities L on two simple DAGs

node named a2 in Figure 3b is clearly distinguishable from that named a1 simply by comparing

the locally stored lists, L, of names which refer to a common internal object.

In general, there may be several such name lists L on a node with complicated descendants. We assumed in Comparator 1 Steps 3 and 4 that the children of all objects in the same formal class are identically indexed by J. Consider again the DAG in Figure 3. In Figure 4 we represent the list of



L1: (Reactor.Output; Stream1; Flash.Input)

OR

L2: (1.1; 2; 3.1)

Figure 4 Representing a name list as a child index list.

names (the subgraph identity) for the bottom node of the DAG as seen from the top node with a list of names (L1) or an equivalent list (L2) of lists of integer child indices. Comparator 2 defines a method for comparing sets of subgraph identities stored as L2 on formally similar objects. This comparator necessarily is transitive and similar to Comparator 1 in returning -1 or 1 if the compared objects have different subgraph identity lists (and hence different deep structures) or 0 if the objects have identical DAG structures.

Comparator 2: Comparing subgraph identities for two objects Oa and Ob of identical formal class

Get SGIList1, the list of subgraph identities stored on Oa.
Get SGIList2, the list of subgraph identities stored on Ob.
If SGIList1.length < SGIList2.length return 1.
If SGIList1.length > SGIList2.length return -1.
For i in 1 to SGIList1.length
    Get names list Identity1, the ith element of SGIList1. (Identity1 is a list like L2 in Figure 4.)
    Get names list Identity2, the ith element of SGIList2.
    Get N1, the number of names in Identity1. (For L2 in Figure 4, N1 would be 3.)
    Get N2, the number of names in Identity 2.
    If N1 < N2 return 1.
    If N1 > N2 return -1.
    For j in 1 to N1
        Get Name1, the jth name in Identity1. (Name1 is a list of integers, such as 1.1 in L2 of Figure 4.)

```
                    Get Name2, the jth name in Identity2.
                    Get P1, the length of Name1.
                    Get P2, the length of Name2.
                    If P1 < P2 return 1.
                    If P1 > P2 return -1.
                    For k in 1 to P1
                            Get C1, the kth element (a child index) of Name1.
                            Get C2, the kth element of Name2.
                            If C1 < C2 return 1.
                            If C1 > C2 return -1.
                            Next k.
                    Next j.
            Next i.
    return 0.
```

Permutations on Comparator 2, such as checking the lengths of all lists in a set of lists for equality before executing element-wise content comparisons, are possible so long as the transitivity property of the comparator is preserved. All elements of the list of subgraph identities and the names within each of these elements must also be stored according to some arbitrary but consistent sort order so that the loops indexed by *i*, *j*, and *k* in Comparator 2 will compare corresponding members of each list. We shall see that this order can be obtained by construction without sorting in Algorithm 2 which we now introduce.

Nonredundancy in computing subgraph identities

We have seen how to compare and make explicit the anonymous class of objects in a DAG. When all the anonymous types of the children of a pair of nodes are identical and we are forced to compare the deep structures, we want to do so with a minimum amount of work. In particular we wish to avoid computing and storing on each node the union of *all* subgraph identities contained in the node's subgraph as this union of identities would essentially duplicate the subgraph. We can avoid such redundancy by noticing that the anonymous classification of each child node, C, of parent node P already accounts for all the subgraph identities which can be expressed using only the name in P which refers to C. Figure 5 gives the simplest example, showing that no identities need

to be recorded on the top node.

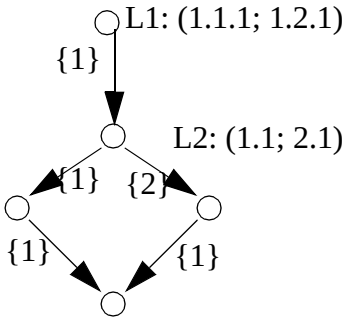


Figure 5 Top node subgraph identity L1 is redundant with L2

Thus, we need record on P only those subgraph identities which are not implied by the combined anonymous classes of its children.

We present a less trivial example in Figure 6, where we show the complete, nonredundant list of subgraph identities on each node. For ease of reference we add a bottom-up numbering of the

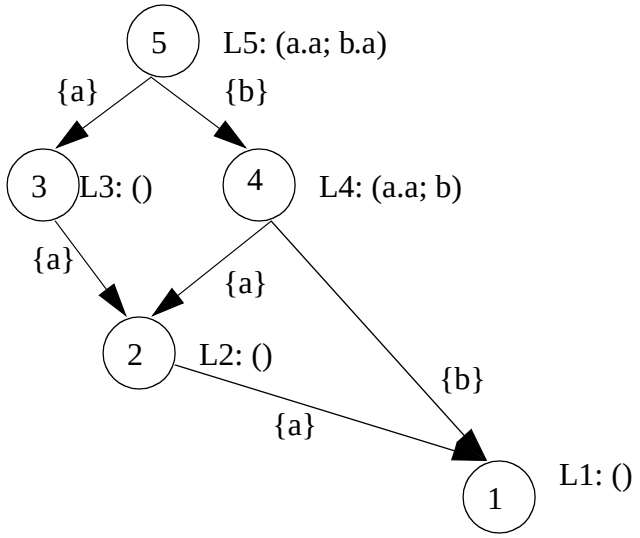


Figure 6 Nonredundant subgraph identities example

nodes as well as the child indexing shown in braces on the edges. The list of identities L5 does not contain the redundant identity R:(a.a.a; b.a.a; b.b), the three ways of naming node 1 starting at node 5. Why? The last two elements (b.a.a; b.b) of this redundant identity repeat the identity recorded on node 4 with the prefix “b.” added, so only one of these can possibly be needed. Let us take the first one of these, b.a.a, and consider the first two elements of the redundant identity:

(a.a.a; b.a.a). This pair repeats the identity shown on node 5 with the suffix ".a" added. The node 4 and its descendant node 2 together form a supernode which contains all the links to node 1. Identities of node 1 are of no concern to nodes above this supernode.

We can more easily see the redundancy and eliminate it using the following graphical algorithm. This algorithm eliminates redundant identities for a node m found among the descendants of node j by considering only the two sets of nodes: j with its children and m with its parents. The connectivity of other intermediate nodes in the DAG is unrepresented except as lists of reachable nodes stored on the children of j.

We now present Algorithm 2, a constructive proof that the minimum nonredundant set of subgraph identities needed to account for the interconnections in a formally typed DAG can be found. In example 1 of Algorithm 2 which accompanies it, we shall show how (a.a.a; b.b) is also a redundant identity for node 1 as seen from node 5; there was no sleight of hand involved in choosing b.a.a instead of b.b from among the elements of R to consider for redundancy with a.a.a. The graph of Figure 6 is embedded in Figure 7 as nodes 2, 3, 4, 5, and 8.

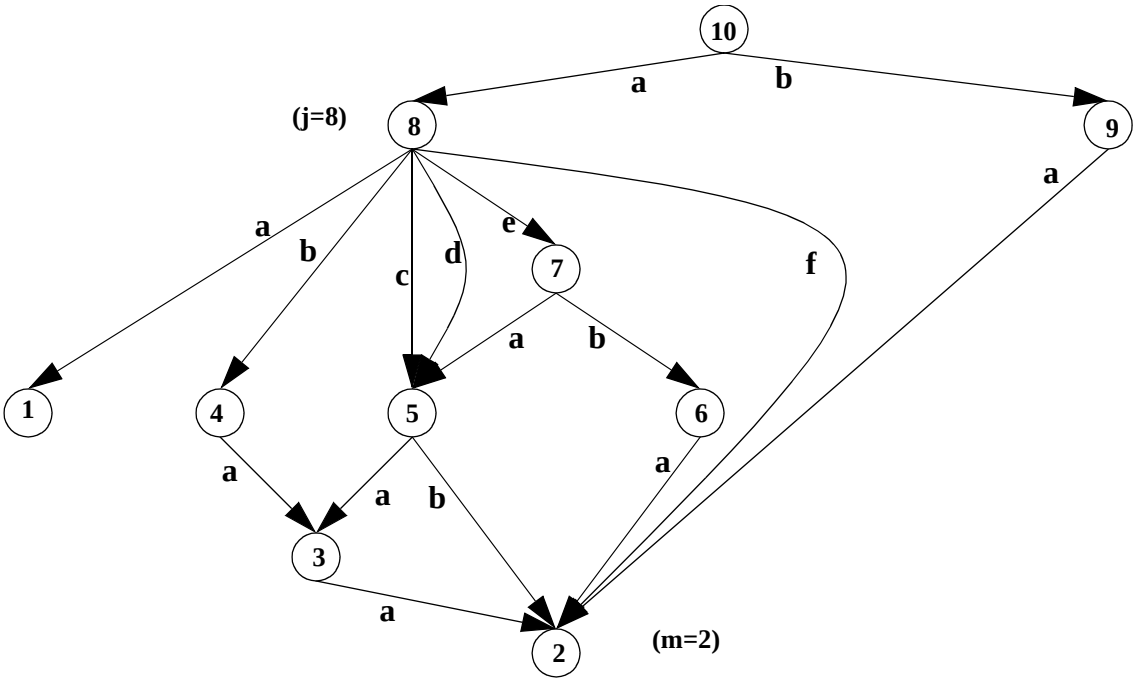


Figure 7 Partial DAG for Example 1

Algorithm 2: Graph derivation of nonredundant identities in a formally typed DAG.

1 *Index the nodes of the DAG in a depth-first, bottom-up sequence.*

2 *Compute and store on each node k in the DAG the list of indices of all nodes reachable from k, D^k, including k in this list.*

3 *To compute the identities which must be stored on a node j to account for any sharing of a descendant node m that is not accounted for by the anonymous class of the children of node j, proceed as follows for each combination of j and m in the DAG. For this example, take node j to be node 8 of Figure 7 and m to be node 2.*

4 *Draw a tree T of j and its children with each edge leaving node j going to a separate node as shown in Figure 8A. Delete from tree T every child node k such that m does not appear in the list of descendants of k, D^k, computed in step 2. Also delete the edges leading to deleted child nodes to yield Figure 9 Tree T.*

5 *If there is only one edge left in tree T, stop. This child of T has an anonymous type which accounts for all identities of m in j.*

6 *Draw a tree B of m and its parent nodes with each parent having only one edge leading to m, as shown in Figure 8B. Delete from tree B every parent node k such that k = j or such that k does not appear in the list of descendants of j, D^j, computed in step 2. Also delete the edges leading to m from deleted parent nodes to yield Figure 9 Tree B.*

7 *From each child node k remaining in T, draw an edge to every parent node p in B such that p is in the descendants of k list D^k. Figure 9 shows with the new edges included as dashed lines the result of this step applied to Figure 8. The resulting DAG TB summarizes all possible identities for node m in the scope of node j since all such identities may be expressed completely as either an edge in T (e.g. edge f) or a path starting with one of the edges in T and ending in one of the edges in B. We have lumped together the irrelevant details of intermediate nodes into the dashed edges.*

8 *For each parent p of node m in DAG TB, merge the parent nodes of p among the children of j into a supernode, as shown in Figure 10, merging all edges entering the supernode but keeping a list of*

original edge names.

9 *If only one edge remains leaving node j, stop. The anonymous classification of one or more children of j accounts for every possible identity of m.*

10 *For each child k of node j in DAG TB, merge the children of k among the parents of node m into a supernode.*

11 *If any nodes were merged in step 10, go to step 8.*

12 *For each edge leaving node j in the final graph TB (Figure 11), find and add to list* L^{jm} *a complete path to node m through the original DAG of Figure 7. Each path must be traced in an arbitrary but consistent way so that comparisons may be made later with Comparator 2. We take the left most edge when any supernode of TB or node of G presents a choice of edges. The result is list L8: b.a.a; f.*

13 *Add the list of identities computed in step 12 to the list of identity lists stored on node j.*

The reader may verify understanding the algorithm by computing the other identities which must be stored for node 8 on Figure 7: (node 3- b.a; c.a. node 5- c; d; e.a.)

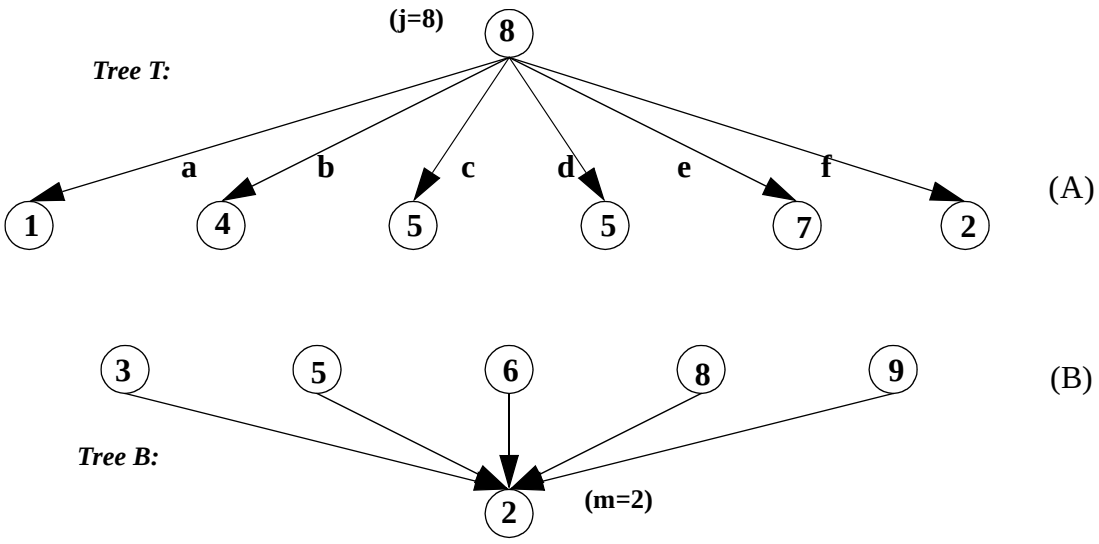


Figure 8 Children of j = 8 (T) and parents of m = 2 (B).

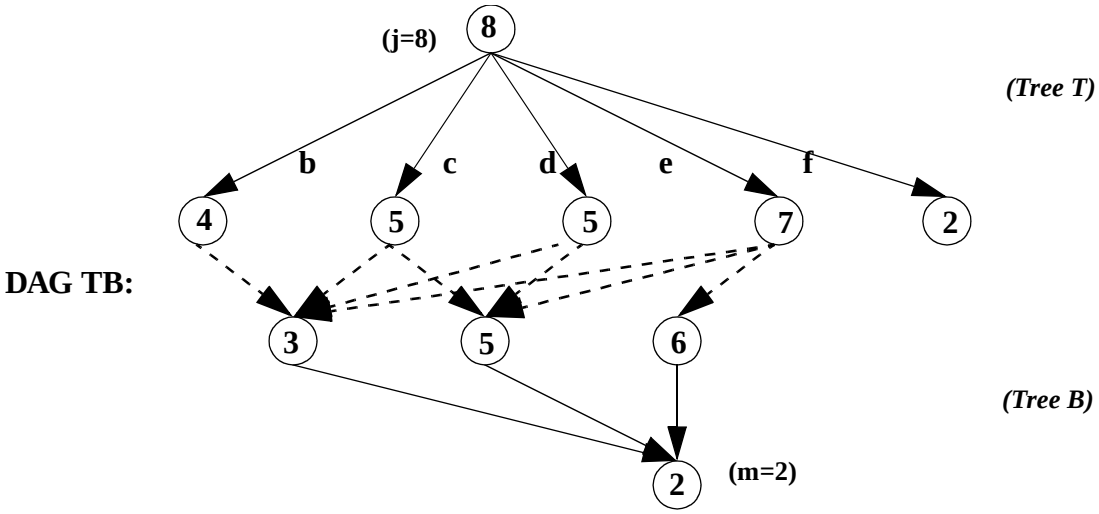


Figure 9 DAG TB for detecting redundancy

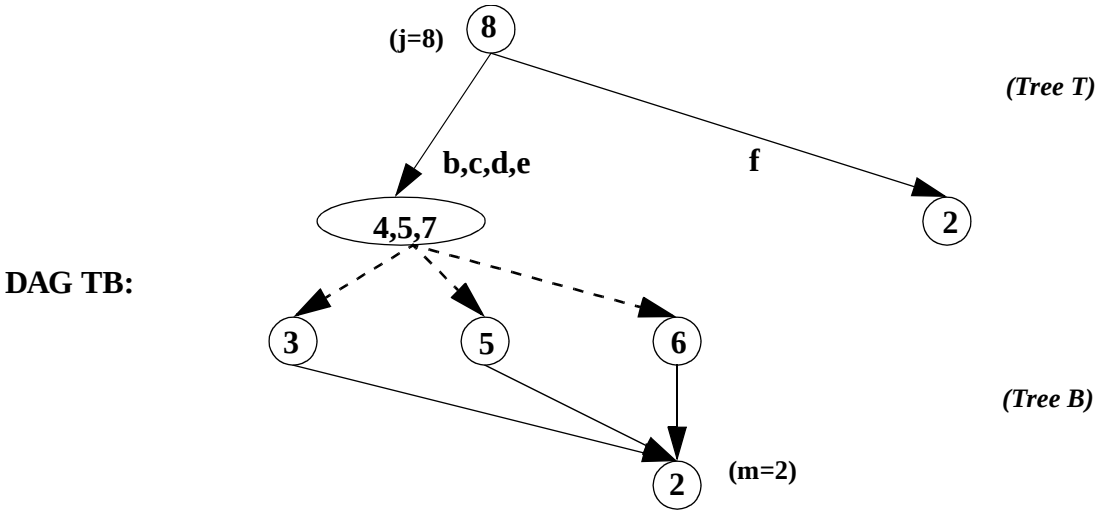


Figure 10 DAG TB for detecting redundancy

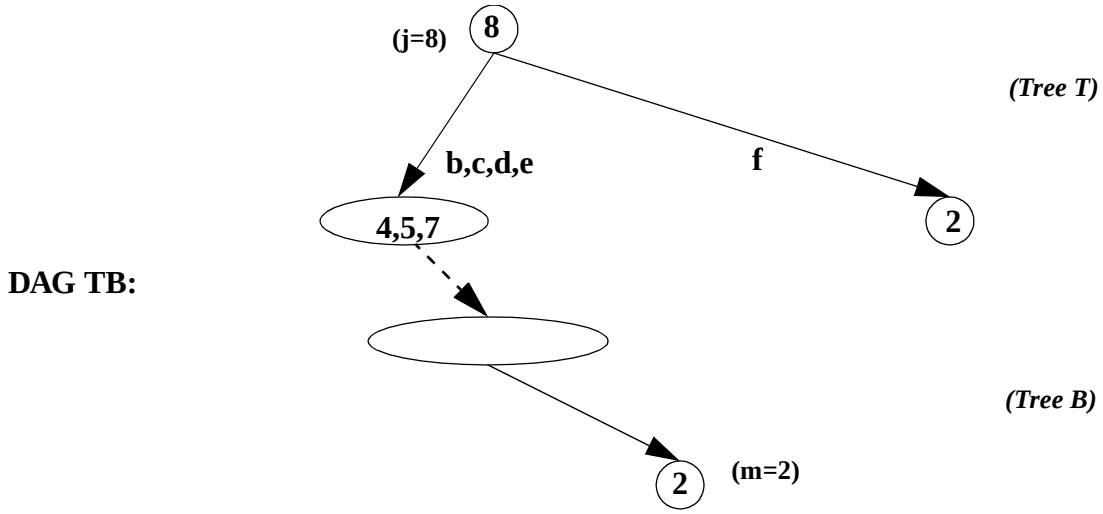


Figure 11 DAG TB for detecting redundancy

Cost of algorithms

The algorithms and comparators just described are polynomial in the number of nodes or edges in the DAG being processed. Algorithm 2 is the most complex and, as given in graphical form, might seem to be factorial in the number of nodes in the DAG, since we might have to apply it pair-wise to each descendant of every node in the graph. Using the compressed index lists described, Algorithm 3 of the Appendix describes an implementation of Algorithm 2 which is polynomial in the number of nodes of the processed graph. In practice Algorithm 3 is nearly linear in cost, as seen in Table 5. We expect this because DAGs which describe physically based models typically have many islands of highly interconnected nodes with relatively sparse interconnections between these islands.

[tables 4,5 about here]

Cost of algorithm 1:

Algorithm 1 steps 1 to 4 are each O(1), and contribute O(NN) when executed over all NN nodes.

Algorithm 1.1 applied to node j requires a binary search of the ACList corresponding to the formal class of j. Potentially all j-1 already sorted nodes are in this same ACList, so this step could require O(log(j)) applications of Comparator 1. Over all NN nodes, algorithm 1.1 could require at

worst $\sum_{j=1}^{NN} \log j$ comparisons. This worst case is very seldom obtained in physically based models where independent of j there are typically only two or three entries in any ACList; thus, we expect to conduct only O(NN) comparisons.

We must still account for the cost of the Comparator 1 at node j. Steps 1 and 2 of the Comparator 1 are O(1). Steps 3 and 4 require comparing the anonymous classes of corresponding child nodes of node j and the exemplar object E. Potentially there are j-1 such O(1) child class comparisons, yielding O(j) cost. In addition, if no difference is yet found we must also compare the subgraph identities stored on j and on E. Thus far we have the cost of algorithm 1 as $\sum_{j=1}^{NN} \log(j)(O(j) + O(\text{comparator 2}))$. Comparator 2 requires at worst a three level iteration. For the outer iteration there can be no more than j-1 identities stored in the SGIList of j since we cannot have more identities than we have descendants of node j. Because we record only non-redundant names in an identity, there can be at most j-2 names in an identity to be processed in the middle iteration. In the inner loop, there can be at most j-1 elements in a name. Thus we have $O(j^3)$ integer comparisons to perform in Comparator 2.

We now have a complete expression for the cost of Algorithm 1: $\sum_{j=1}^{NN} \log(j)(O(j) + O(j^3))$ which is $O(NN^4 \log(NN))$. We do not expect to see this order in practice. The inner loop of Comparator 2 is bounded by the maximum depth of the DAG which is typically a small number such as ten in a physically based model. This reduces the j^3 term to j^2. We also expect many of the NN objects to fall in the atomic and constant categories of Steps 1 and 2, so that for most j only the O(j) term of Comparator 2 applies. Finally, because most physical models are sparsely interconnected and are composed of container classes which do not possess a descendant of the same container class we expect the $j \bullet \log(j)$ of $\sum_{j=1}^{NN} j \bullet \log(j)$ to reduce to some constant term dependent on the physics and modeling style in use. In Table 4 we see that algorithm 1 time is linear in the number of nodes, with different coefficients for the C model series and the SREC model series.

Cost of algorithm 3:

Algorithm 3 is Algorithm 2 recast in more readily computable terms, Steps 1 ,3, 4, 5, and 9 are easily seen to require O(NN) operations. Steps 2, 6, and 7 are easily seen to require $O\left(\sum_{j=1}^{NN} Nc^j\right) = O(NE)$ operations.

Steps 8 and 9, constructing the node reachability lists S^j and D^j is done in a depth-first, bottom-up sweep through the DAG which computes at each node the union, S^j or D^j, of the corresponding reachability lists of its child nodes. This step dominates the cost of Algorithm 3, as seen in Table 4, though it is not the highest order step. For node j with Nc^j children this union of sorted lists has a maximum cost of (Nc^j * cardinality(D^j)). Over the entire graph we have $\sum_{j=1}^{NN}\sum_{k=1}^{Nc^j} cardinality(D^k) < \sum_{j=1}^{NN}\sum_{k=1}^{Nc^j} NN = NN \bullet \sum_{j=1}^{NN}\sum_{k=1}^{Nc^j} 1 = NN \bullet NE$, where NE is the number of edges in the graph. Step 9 therefore is at worst O($NN \bullet NE$) in operations for naive list storage. The compressed list storage which is necessary to avoid O(NN^2) memory cost also in practice reduces this O($NN \bullet NE$) term to O(NN) because the compressed lists will not exceed some relatively small length which is dependent on the modeler's style of decomposing the physical system.

Since Step 11 is not cost dominant in practice we will not derive its order in detail. The complex loops are O(NN^2NE) in the worst case but linear in practice. The final element of step 11, tracing and recording the paths that make up each identity, is O(NN^3log(NN)*Depth of DAG) < O(NN^4log(NN)). Starting at a child node of j which we know to be the start of an identity ending at node m, we descend through the DAG choosing at each node entered to follow a leaving edge such that the reachability list of the node at the other end of the edge contains m. Potentially this descent could require Nc^k list searches (log(k) cost each) at each intermediate node k. Summed over all nodes j, all shared descendants m of j, all possible children of j (Nc^j < NN), and the maximum path length (Depth of DAG < NN) we obtain O(NN^4log(NN)) operations.

In practice, the depth of the DAG is usually limited to a small number rather than NN, we do not find and record redundant paths, and we seldom need to search over all children of a node to select

a leaving edge, so we expect this step to be roughly linear in cost. As already noted, storing node reachability information in sorted, compressed lists of integers is necessary to avoid NN^2 storage costs. One-of-a-kind objects will never be compared, so detection of subgraph identities should not be performed for these objects. Incremental search of reachability information, not apparent in Algorithm 2, is seen in Algorithm 3 step 11. In practice this reduces quadratic performance of Algorithm 3 to linear.

Conclusions

The structures that naturally result from a user-oriented process model description make very high performance obtainable without sacrificing the expressive power and flexibility [11] of equation-based languages. We have explained the issue of anonymous class detection in process simulation DAG structures. We suggest the ways in which anonymous class detection can transform today's equation-based modeling systems with large object-oriented overhead memory and CPU costs suitable only for small modeling projects into scalable systems capable of performing on the same level as hand-coded binary library flowsheeting systems. Based on promising early results presented here, we will continue to investigate anonymous class exploitation. Engineers can look forward to future large-scale flowsheeting systems with the richness of features available from today's small-scale highly interactive, object-oriented, equation-based systems.

Acknowledgments

The member companies of the Computer Aided Process Design Consortium and the National Science Foundation through its grant, No. EEC-8943164, to the Engineering Design Research Center provided the financial support of this project.

Nomenclature

a.b notation for a whole-part relationship, b is a part of a, as commonly seen instructured modeling and programming languages. Also notation for a sequence of edges (a, b) and nodes (.).

A(k,q) element in the kth row and qth column of a sparse matrix.

C or c child node in a DAG or the numeric index assigned to a child node.

C^k children of k; the nodes reachable by traversing each edge leaving node k.

Csi^j the list of children of node j which have a share index SI > 0.

Cni^j the list of children of node j which have an interior index NI > 0.

D^k list of all nodes reachable from node k by following directed paths of any length leaving k. The zero length path is included, so k is in D^k.

F the formal class, existing in a class library.

j, k, m three nodes in a DAG or the numeric indices of those three nodes.

L^{jm} a list of edge-node paths leaving node j and terminating on node m.

Nc^j Number of edges leaving node j.

NE Total number of edges in a DAG. $NE = \sum_{j=1}^{NN} Nc^j$.

N^i global index of the ith node in a DAG where every node is indexed and every node is reachable from the top node $NN(N^N)$.

NI^k interior index of the node with global index k. Nodes without shared descendants have NI zero.

NN total number of nodes in a DAG and the global index of the topmost node.

Np^j number of edges entering a node j from unique parent nodes.

Ns^k total number of shared nodes j among the nodes descendant from k. Shared nodes are those which have $Np^j > 1$.

O, Oa, Ob a compiled object, an instance, which has an associated node in a DAG representation.

P or p parent node in a DAG or the numeric index assigned to that parent node.

P^k parents of k; the nodes reachable by traversing backward each edge entering node k.

P^{jm} The list of nodes which are both descendants of node j and parents of node m. j may be a member of this list but m may not.

S^k list of all share indexed nodes (SI > 0) found among nodes descendant from the node with global index k.

SI^k share index of the node with global index k. Nodes with only one parent have SI zero.

x[i] ith element of a vector or list x.

Bibliography

[1] Kirk Andre Abbott. *Very Large Scale Modeling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1996.

[2] Benjamin A Allan, Vicente Rico-Ramirez, Mark Thomas, Kenneth Tyner, and Arthur Westerberg. “ASCEND IV: A portable mathematical modeling environment.” ICES technical report, number not yet assigned, October 6 1997. available via http://www.cs.cmu.edu/~ascend/ascend-help-BOOK-21.pdf.

[3] Benjamin Andrew Allan. *A More Reusable Modeling System*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA USA, April 1997.

[4] Mats Andersson. *OMOLA - An Object-Oriented Modelling Language*. PhD thesis, Lund Institute of Technology, Department of Automatic Control, Lund, Sweden, 1990.

[5] Paul I. Barton. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, London, 1992.

[6] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS - A user’s guide, Release 2.25*. Scientific Press, 1992.

[7] Thomas Guthrie Epperly. “Implementation of an ASCEND interpreter.” Technical report, Engineering Design Research Center, Carnegie Mellon University, 1989.

[8] D. M. Gay. “Hooking your solver to ampl.” Numerical Analysis Manuscript 93-10, AT&T Bell Laboratories, August 1993.

[9] Wolfgang Marquardt. “An object-oriented representation of structured process models.” *Computers and Chemical Engineering*, 16S:S329–S336, 1992.

[10] Peter Colin Piela. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1989.

[11] Arthur W. Westerberg and Dean R. Benjamin. “Thoughts on a future equation-oriented flowsheeting system.” *Computers and Chemical Engineering*, 9(5):517–526, 1985.

Appendix

Algorithm 3: Deriving minimal Subgraph-Identities (information about node sharing in a DAG) to detect deep structural differences during anonymous classification

Unless otherwise noted, all lists in this algorithm are lists of integers sorted in increasing order.

1 *Count the nodes (instances) of each formal class and the total number of nodes.*

2 *Count the number of child links coming into each node, j, and record it as* Np^j.

3 *Mark nodes which have a unique formal class (so we can ignore them later).*

4 *Apply a depth first, bottom-up numbering, starting from 1, to all nodes j such that* $Np^j > 1$ *and j is not marked as the unique instance of a formal class. These are the shared nodes we must account for when computing subgraph identities, and the number applied to each j is its share-index,* SI^j. *Assign all nodes which do not qualify a share-index of 0. (The total number of shared nodes is* Ns^{NN}.*)*

5 *Apply a depth first, bottom-up numbering, starting from 1, to all nodes j such that* $Np^j > 1$ *or j is an instance of a container class or j is an instance of an array class. These are the nodes which may need subgraph identities recorded on them, and the number applied to each j is its node-index,* NI^j. *Assign all nodes which do not qualify a node-index of 0. (This step, in effect, trims off those leaves of the DAG which are of strictly local interest.)*

6 *Collect and store on each node, j, the list of its child nodes k such that* $SI^k > 0$. *This list is* Csi^j, *and it is a list of pointers to nodes rather than a list of integers.*

7 *Collect and store on each node, j, the list of its child nodes k such that* $NI^k > 0$. *This list is* Cni^j, *and it is a list of pointers to nodes rather than a list of integers..*

8 *Collect and store on each node, j, the list of non-zero share-indices found on node j or any of its descendant nodes. This list is* S^j.

9 *Collect and store on each node, j, the list of non-zero node-indices found on node j or any of its descendant nodes. This list is* D^j.

10 *Create a contiguous array, S2N, which maps the share-index of each shared node to its node-index.* $S2N[SI^k] = NI^k$.

11 *For each node j in the DAG*

If length of $Csi^j < 2$ *then continue with next j.*
Set LS to the empty list.
Collect S^k *from each node k in* Csi^j *and add it to LS, making a list of lists.*
Set LD to the empty list.
Collect D^k *from each node k in* Cni^j *and add it to LD, making a list of lists.*
For k in 1 through NLS
Set PositionHint[k] <-- length of kth list in LS.
Next k
For each share-index m in list S^j *in reverse order*
If m == j then continue with next lower m.
Get NLS, the length of list LS.
Set RootNode-Index <-- S2N[m].
Set StartingPointsList of lists to the empty list.
For k in 1 through NLS
While PositionHint[k]th element of kth list in LS > m
Decrement PositionHint[k].
EndWhile

```
        If PositionHint[k]th element of kth list in LS == m then
            Add the kth list in LS to starting points list SP.
        Next k.
If length of StartingPointsList < 2 then
        Next m.
Set P^jm <-- D^j INTERSECTION P^m.
For each k in SP
        For each q in P^jm INTERSECTION D^k
            Create element A(k,q) in sparse matrix A.
            Increment ColumnCount[q].
            Next q.
        Next k.
For each q in P^jm
        If ColumnCount[q] is zero then
            Next q.
            # m is a direct child of j, and this q is j.
        If ColumnCount[q] is one then
            k = row index of the single matrix element A(k,q).
            If Group[k] is unassigned, then
                Assign Group[k] = k.
                Add Group[k] to GroupList[k].
            Next q.
        # ColumnCount[q] > 1
        Unassign LargestGroup
        Assign LargestSize = 0
        Assign GroupsInColumnList = Empty
        Assign NewKInColumnList = Empty
        For each element A(k,q) in column q
            k = row index of element A(k,q).
            If Group[k] unassigned then
                Add k to NewKInColumnList
            Else
                If Collected[k] not TRUE then
                    If GroupList[Group[k]].size > LargestSize then
                        LargestGroup = GroupList[Group[k]].
                        LargestSize = LargestGroup.size.
                    Add GroupList[Group[k]] to GroupsInColumnList
                    Assign Collected[k] TRUE
            Next element
        If GroupsInColumnList is Empty then
            Assign KNewGroup = last entry of NewKInColumnList
            For each k in NewKInColumnList
                Assign Group[k] =KNewGroup
                Add k to GroupList[KNewGroup]
                Next k
        Else
            Assign KOldGroup = LargestGroup
            Merge the groups of [GroupsInColumnList less LargestGroup] with Largest-
              Group
            Assign Collected[KOldGroup] = FALSE
            For each k in NewKInColumnList
                Add k to GroupList[KOldGroup]
                Assign Group[k] = KOldGroup
```

```
                                    Next k
                        Next q
                Assign Collected[k] FALSE for all k in SP.
                Clear matrix A for use with next m.
                Assign final path list, FP, empty.
                For each k in SP
                        If Group[k] unassigned then
                                #direct link from j to m
                                Add k to final path list, FP.
                                Assign Collected[k] = TRUE
                        Else
                                If Collected[Group[k]] not TRUE then
                                        Assign Collected[Group[k]] = TRUE
                                        Add k to list FP.
                        Next k
                If Length of list FP < 2 then
                        Next m
                Else
                        Apply Algorithm 4 to find a route, Path^K, from node j to node m starting with each
                            child link, K, in the list FP.
                        Save all Path^K on node j as a tuple describing shared node m.
                        Next m
                Next j
Stop.
```

Algorithm 4: Finding a canonical path from from node j to node m.

Given root node j, shared index, SI^m , of target node m, and starting child index (*start*) in node j of the path,

```
1   Init Path, the list of child indices we seek, to the empty list.
2   Init nextindex to start.
3   Init parent to node j.
4   Init boolean Found to false.
5   While Not Found
                Init boolean KeepEdge false.
                While Not KeepEdge
                        Get child from edge nextindex in parent.
                        Get reachability List from child .
                        If SI^m m in List then
                                Set KeepEdge true.
                                If shared index of child equals SI^m
                                        Set Found true.
                                EndIf
                                Append nextindex to Path.
                        Endif
                        Increment nextindex.
                EndWhile
                Set nextindex to 1.
```

Set parent to child.

EndWhile

6 Stop.

Comment: Path now contains the route from j to m

Table 4: Classification algorithm and equation construction timing

Model	Total	Eqns	Classification	Identity Detection			total nodes	shared nodes	identities
	sec	sec	sec	total sec	reachability calculation	identity discovery			
srec	2.0	1.2	0.2	0.5	0.4	0.1	1038	359	948
srec2	2.9	1.4	0.3	1.1	0.7	0.3	2060	718	1896
srec3	3.7	1.6	0.4	1.6	1.1	0.3	3081	1077	2844
srec4	5.1	2.2	0.6	2.2	1.5	0.4	4102	1436	3792
srec5	6.8	3.3	0.7	2.6	1.8	0.5	5123	1795	4740
srec10	10.2	2.9	1.4	5.6	3.7	1.2	10230	3590	9480
srec20	19.6	5.2	2.7	11.2	7.4	2.5	20444	7180	18960
srec40	48.5	18.4	6.2	22.9	15.2	5.2	40872	14360	37920
srec80	82.0	17.9	14.3	48.0	31.5	11.4	81728	28720	75840
c6	1.2	0.9	0.1	0.1	0.1	0.1	456	147	277
c7	1.2	0.9	0.1	0.2	0.1	0.0	504	166	320
c8	1.3	0.9	0.1	0.2	0.1	0.0	552	185	363
c16	1.8	1.1	0.1	0.5	0.3	0.2	936	337	707
c32	2.3	1.2	0.2	0.9	0.5	0.2	1704	641	1395
c64	3.9	1.5	0.4	1.9	1.2	0.4	3240	1249	2771
c128	7.4	2.0	0.8	4.4	3.1	0.8	6312	2465	5523
c256	17.6	3.3	1.7	12.3	9.8	1.7	12456	4897	11027
c512	51.0	7.0	3.7	39.6	33.9	4.1	24744	9761	22035

Table 5: Compile time reduction

Model	Nuc	Nus	Neqn	Mbyte		CPU		phase 1		phase 2		wall clock	
				new	old	new	old	new	old	new	old	new	old
srec	310	71	1609.	1.7	2.7	4.6	10.3	2.6	2.6	2.0	7.7	9	13
srec2	313	71	3218.	3.2	5.3	8.3	20.5	5.2	5.0	2.9	15.4	9	22
srec3	313	71	4827.	4.5	7.8	11.6	30.8	7.7	7.6	3.7	23.0	12	31
srec4	313	71	6436.	5.8	10.3	15.5	41.1	10.2	10.3	5.1	30.6	16	43
srec5	313	71	8045.	7.1	12.8	20.1	52.2	13.0	12.7	6.8	39.1	21	53
srec10	313	71	16090.	13.3	25.2	39.7	103.4	28.9	25.9	10.2	76.9	41	105
srec20	313	71	32180.	26.0	49.8	89.9	209.0	69.1	52.9	19.6	154.9	94	212
srec40	313	71	64360.	49.8	99.2	157.1	420.1	106.0	106.2	48.5	311.5	173	426
srec80	313	71	128720.	97.1	195.2	322.4	964.8	235.7	269.0	82.0	653.7	341	1377
c6	229	71	671.	1.0	1.3	2.4	4.3	1.1	1.1	1.2	3.2	14	19
c7	229	71	760.	1.1	1.5	2.4	4.8	1.2	1.2	1.2	3.6	3	5
c8	229	71	849.	1.2	1.6	2.6	5.5	1.3	1.3	1.3	4.1	3	6
c16	229	71	1561.	1.8	2.7	4.2	9.7	2.4	2.4	1.8	7.3	5	11
c32	229	71	2985.	2.8	4.9	7.0	18.3	4.6	4.3	2.3	13.9	8	18
c64	229	71	5833.	5.0	9.1	13.0	36.0	8.9	8.6	3.9	27.2	14	38
c128	229	71	11529.	9.3	17.8	25.9	71.5	18.2	17.3	7.4	53.7	32	78
c256	229	71	22921.	17.8	34.9	58.8	143.1	40.4	35.4	17.6	106.9	64	148
c512	229	71	45705.	34.3	69.1	132.5	295.8	79.8	78.6	51.0	215.5	137	307