Reusability and Scalability of Models

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

Abstract

Object-oriented, equation-based modeling systems (ASCEND, Omola, gPROMS, VeDa and others) 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 or modification of libraries. We present declarative modeling formalisms which aid the library user both in correct application and correct modification of complex library models.

Compile times become an issue 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 progress in developing automatic compilation algorithms which reduce times from minutes to seconds by taking into account the rich hierarchical semantics of user-written object definitions.

We find that many of the same formalisms which aid in model application and modification also significantly improve compiler speed and diagnostic services. We also find that discovery of the anonymous types of compiled objects allows for large reductions in CPU time and core memory usage. Preliminary results indicate that interactive, interpreted modeling systems may soon become as fast as less flexible batch systems which rely on very rich libraries of precompiled binary modules.

Introduction

Equation-based modeling systems have been investigated as tools to improve the efficiency of the modeling process, particularly in the uncharted situations routinely faced by process researchers and process designers. We illustrate a common modeling process in Figure 1. This process can describe nearly any form of modeling or research. In this paper, we focus on the creation and use

of computational models, a significant piece of "overall modeling" for many chemical engineers.

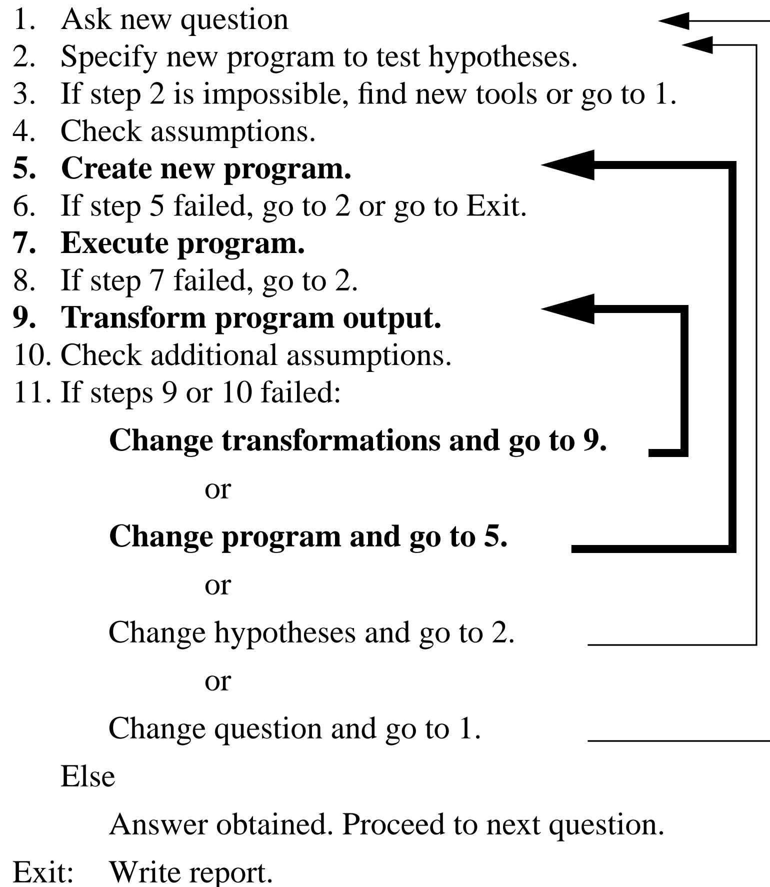

Figure 1 An iterative modeling process.

Interactive equation-based systems improve the efficiency of the steps shown in bold by:

- interpreting object-oriented and hierarchical model description languages at interactive speeds. This eliminates the compile-link-run cycle of traditional programming languages.

- providing exact gradients. This eliminates the CPU requirements and numerical errors of computing finite difference gradients, a task which can significantly slow simulation and optimization algorithms.

- allowing many models to be represented by one compiled object which can be interactively

configured in many ways by setting flags. This eliminates compiling a separate model for each configuration.

- expressing flowsheet superstructure representations, which a solution engine can manipulate automatically in an optimization process. This eliminates time spent performing manual case studies.

- allowing the user to apply general purpose solvers or model reformulation tools for distinct problem classes to the same model representation. This eliminates writing a different program to solve each problem. For example, creating an initial value formulation of a differential-algebraic system just to initialize a separate boundary value problem is not necessary.

- allowing the user interactive access to all sub-models, so that each part can explored in the process of discovering overall initialization and solution strategies. This eliminates compiling separate, small models when subproblems prove difficult or behave surprisingly in the larger context, as they frequently will.

Most interactive equation-based systems make gains in one or more of these areas by accepting losses in others. The most common sacrifice (as some regard it) in object-oriented systems is the loss of familiar linear information structuring that is common to spreadsheets and the styles of FORTRAN frequently seen in computing courses for undergraduate chemical engineers. These systems provide few if any hooks for new users from traditional backgrounds to grab during the paradigm shift to declarative, object-oriented modeling. As a result many new modelers simply give up on using advanced modeling tools.

Another example of the loss of familiar language usage occurs in the ASCEND system. The domain-specific iconographic language of flowsheets is not supported because there is no universally accepted mapping to lines and icons from arbitrarily structured hierarchical models written in a general-purpose, text-based language.

In the first section on improving understanding and reuse of models, we introduce a new object passing formalism which offers new users a smoother transition path into declarative modeling and which aids in modification and reuse of model source code libraries. We demonstrate that the same information which helps the user can be used to reduce the time required to compile (instantiate) large objects.

A critical feature of this formalism is that the objects constructed form directed acyclic graphs (DAGs), not simple trees. A similar formalism has been seen as useful in procedural programming languages with varying degrees of object orientation such as C++, Eiffel, and Java. The primary implication of the DAG structure in mathematical modeling systems is that it allows redundant equations to be eliminated easily and it allows independent solution of any part of the overall model. Secondarily, connecting equations, e.g. *`reactor.flow_rate_out = separator.flow_rate_in`*, which can become unwieldy in very large models, are not needed.

In the second section on discovering similar objects, we introduce a general method to regain the speed and compactness of machine code while retaining the flexibility of the interactive systems.

ASCEND III (Piela, 1989), initially sacrificed the extreme speed and compactness of compiled machine code to obtain a flexible representation which supports interactive model respecification and the solution of any arbitrary subproblem. The general method we propose requires determining the *anonymous type* of each compiled object. Models written in reusable libraries generally leave constants which determine the final size of the compiled model 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 model (or class) are of distinct *anonymous type* unless the constants specified to each at compilation time are identical.

The ideas we present are experimentally verified here and elsewhere (Allan, 1997) to allow interactive manipulation of 20,000 or more equations. The experiments have been conducted in the ASCEND IV[1] system (Abbott, 1996, Allan, 1997, Allan, et al, 1997).

Improving understanding and reuse of models is good for users and for compilers

In this section we highlight several modeling language features (Allan, 1997) which we find make it easier to reuse and modify declarative models in any language. The most important of these is a version of object passing which we have not seen described elsewhere in the object-oriented literature. Model reuse and modification are further supported by modeling ideas which were present but not highlighted in ASCEND III (Piela, 1989, Abbott, 1996) and by concepts similar to ideas in the realm of object-oriented software design (Meyer, 1992a). We shall see that application of these features in defining ASCEND IV leads to greater than 60% reduction of compilation times over comparable ASCEND III models, even though significant optimizations in the ASCEND IV implementation remain to be performed. Limited testing of ASCEND IV with undergraduate and graduate students indicates that it is highly reusable. Gains in compilation time help in model reusability by allowing more experiments to be conducted in the time available for modeling activities.

Concept: Object passing

Mathematical models for a piece of chemical process equipment or for a thermodynamic mixture can be quite complicated. Users of any piece of software generally desire clear, simple interfaces, at least until the software in question needs to be used for an application different from the original designer's vision. The user of a mathematical model may need to modify the internal mathematical structure, such as adding reaction terms to the mass and energy balances of a flash unit to obtain a reactor. Even if the model is adequate as received, the interface to the model needs to state concisely all the use requirements so that the user need not read a thick manual which may not even exist. These requirements need to be stated in a way which is understood by the user and easily verified by the compiler in the final application (Meyer, 1992a). We illustrate such an inter-

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

face in Figure 2, using the ASCEND IV syntax.

```
MODEL VL_Flash(
      feed WILL_BE stream;
      n_overheads IS_A integer_constant;
      vapor[1..n_overheads] WILL_BE stream;
      n_bottoms IS_A integer_constant;
      liquid[1..n_bottoms] WILL_BE stream;
) WHERE (
      feed, vapor[1..n_overheads], liquid[1..n_bottoms]
            WILL_NOT_BE_THE_SAME;
      FOR i IN [1..n_overheads] CREATE
            feed.species == vapor[i].species;
      END FOR;
      FOR j IN [1..n_bottoms] CREATE
            feed.species == liquid[j].species;
      END FOR;
);
```

Figure 2 The interface to VL_Flash, a flash unit.

The interface syntax for the VL_flash model type is not extremely different from interfaces in object-oriented procedural languages like Eiffel (Meyer 1992b). We must, however, keep in mind that ASCEND and similar mathematical modeling systems are not procedural, so the semantics of this interface are not precisely those of an Eiffel interface. ASCEND and similar modeling systems do not *hide* substructures, in particular variables may be accessed from any enclosing scope when formulating equations at higher levels. Eiffel and other traditional object-oriented software construction languages use interfaces to hide information.

```
MODEL test_flowsheet;
      feed1 IS_A stream(species);
      offgas[1] IS_A stream(species);
      product[1] IS_A stream(species);
      solvent_degasser IS_A VL_Flash(feed1,1,offgas,1,product);
      species IS_A set OF symbol_constant;
      species :== ['methane','hydrogen_sulfide','heptane'];
END test_flowsheet;
```

Figure 3 A test driver for VL_Flash

In defining the model VL_Flash, we require the user to provide a feed stream and arrays of product streams. The ranges for the arrays are to be specified as constant values. The idiom used to state these requirements is not so different from common procedural languages as to leave new users completely lost. The user does not need to read the equations coded in the body of the model

to determine what the VL_Flash is if the model name and argument names are well chosen. The compiler need not attempt to execute statement 5 in Figure 3 until all the required argument objects and values are available. If the modeling language supports inheritance, then it is reasonable to accept a more refined subclass of the type specified in the model interface, since the more refined subclass furnishes at least the expected attributes of the specified type.

If the user is unfamiliar with the model, writing a test driver is not particularly hard. The driver in Figure 3 is easily derived once one understands that `WILL_BE` means a passed object is required and `IS_A` statements means a constant value is required. This makes it reasonable to construct a simulation and explore it interactively without first reading extensive documentation or tracing through model source code. The primary benefit of software encapsulation, easy reuse, is obtained without frustrating the user who needs to examine or to modify the code in detail at some later time.

It may seem unusual that both feed and product streams must be supplied to a unit model. This illustrates the first of the two sources of savings in compilation time. The streams in and out of the flash unit enter or exit the flowsheet from some hypothetical, unmodeled reservoir or they are shared with another unit in the flowsheet. Any object, such as the stream, which is shared in different contexts should be created once and then be passed to all the sharing contexts. This style of modeling eliminates the lost compiler work caused by creating two streams in separate units and then merging them through some compilation process. The savings are even greater if a shared object is used in many different contexts, as is the case with a model which represents the set of chemical species in use throughout a flowsheet.

The object hierarchies created by passing objects as described above are DAGs. We illustrate this with the code and graph shown in Figure 4. The names specified for parts are shown on the links of Figure 4. The stream exiting the reactor and entering the flash is constructed and passed to both units. The stream or either of the two units can be solved alone or as part of the flowsheet since each model object in the graph can be isolated with all its subtree. The same DAG structure is obtainable in the data structures of any language which has implicit or explicit passing of pointers,

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

Figure 4 (a)

such as FORTRAN, C++, or Java.

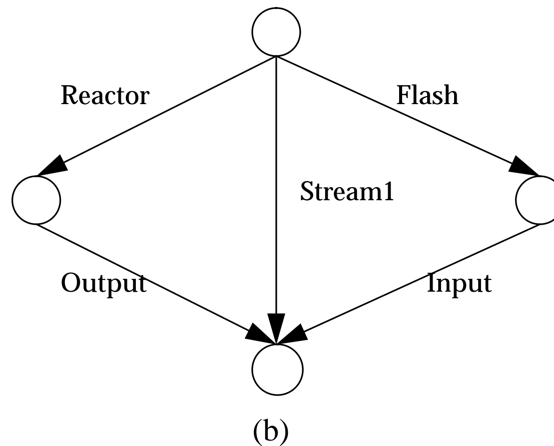

Figure 4 Directed acyclic graph of two units and their connecting stream.

Concept: Argument checking

We now turn to the statements in the WHERE block of the MODEL interface in Figure 2. These statements are conditions on the arguments provided. These are similar to *assertions* in Eiffel programming, but differ in that the conditions can only refer to constant properties of the arguments. Conditions written in terms of variables are disallowed because , though they might be satisfied at compilation time, they can change soon after, thereby leaving an incorrect model. If the arguments provided violate these conditions, then the compiler can issue appropriate warnings and stop. This saves CPU time which might otherwise be wasted on attempting to assemble and diagnose the incorrectly specified data structures. More importantly, this checking gives the user much more precisely targeted warnings and errors, leading to much reduced model debugging time.

This condition checking prevents any misuse of the model that is anticipated by the model author. In our example, the same stream must not be passed to two slots in the interface, as this violates the author's assumptions about mass balance. As end users occasionally report cases of unusual model misuse (which are generally indicative of unstated assumptions in the model), the maintainer of a library model can add additional conditions to prevent future misuse.

Concept: Reusable polymorphism

Polymorphism, the specialization of a basic object definition into subclasses for different particular applications is much discussed in the object-oriented literature. The issue also manifests in mathematical modeling languages. Our experience with ASCEND III has shown that many users, even users with substantial expertise, are wary of creating more refined types (new sub-classes) even when detailed examples of the use of the ASCEND inheritance operators are provided. Users are not confident that they will be able to meet all the unstated assumptions that went into the super-class design and into the models which have been based on existing refinements. Even the graphical type hierarchy display (class browser) added late in the development of ASCEND III

does not make users sufficiently comfortable with their understanding of the thermodynamics library to attempt modifications. Something as simple as adding another vapor pressure correlation put off all our users. It appears that we have overcome some of this reluctance by applying the object passing formalisms already discussed and by creating a clear syntax for selecting among structural alternatives.

```
MODEL liquid_mixture(
      species IS_A set OF symbol_constant;
      mixing_rule IS_A symbol_constant;
) WHERE (
      mixing_rule IN ['Wilson','UNIFAC'] == TRUE;
);

      SELECT (mixing_rule)
      CASE 'Wilson':
            Wilson_mix IS_A Wilson_liquid(species);
      CASE 'UNIFAC':
            UNIFAC_mix IS_A UNIFAC_liquid(species);
      END SELECT;

END liquid_mixture;
```

Figure 5 SELECT code

The declarative SELECT statement syntax illustrated in Figure 5 looks enough like procedural code dispatching to calculation subroutines that a modeler of almost any ability can understand and imitate it. Adding a new liquid mixture correlation becomes a matter of adding another CASE and defining the corresponding model which has the same arguments as the other correlations. The model *liquid_mixture* has no refinements, but it is essentially polymorphic in behavior. There are several liquid correlations which are selected among by the value of *mixing_rule* which is supplied when an instance of *liquid_mixture* is compiled.

This part selection formalism seems more comfortable for many modelers. The examples of other options are all gathered in the containing *liquid_mixture* and the similarity of the interfaces to *Wilson_liquid* and *UNIFAC_liquid* strongly suggests that the same objects should be passed to any recipe for calculating mixture properties. In addition, this formalism allows the compiler to construct each object using its final form rather than constructing an object in superclass form then later expanding it according to the subclass finally selected.

Similarly, a formalism (general functions, (Westerberg and Benjamin, 1985)) for representing solution-time model structure alternatives less restrictive than substitution of one equation for another equation in the same variables helps users express dynamic polymorphism. The ASCEND syntax for general functions (the WHEN statement) is described more fully elsewhere (Ramirez et al, 1997, Zaher, 1995,). General functions allow the user to combine several related models into one and switch among them interactively.

For example, a column model can be compiled with rigorous thermodynamics and full energy

balance equations included. If this column is modeled using general functions that dynamically turn off and on enthalpy models and equilibrium equations, then the user can select dynamically among mass-balance, mass-balance with energy balance errors calculated, and full energy balance. The general function formalism can also be mapped automatically to various discrete optimization formulations (Ramirez et al, 1997). Again, this dynamic polymorphism saves the user time by avoiding recompilation of detailed models after simple ones are first explored.

Concept: Graphical presentation of the object passing formalism

The graphical presentation of interconnections among models frequently provides insight into their proper manipulation and expected behavior. Abstract line-and-box representations are common to virtually every engineering discipline. Unfortunately, workers in any two fields seldom agree on just what a line and a box represent. One approach we are investigating is illustrated in Figure 6. We map all objects as icons, and we show *connections* (common parts passed through interfaces) as small boxes on the edges of icons. Each edge box is connected by a line to the icon of the object being passed. If more detail about a given icon is required, its box can be exploded interactively to show another level of complexity.

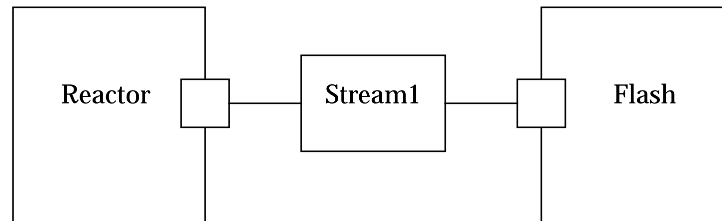

Figure 6 A lines and boxes representation of object passing corresponding to the DAG of Figure 4.

Presently we are testing this representation on paper and blackboard rather than in software. This graphical representation needs to be similar enough to the diverse engineering drawing conventions that an engineer from any discipline can quickly connect it to his or her particular conventions. At the same time, this graphic presentation must be sufficiently different from the particular conventions that the semantics do not become confused among them. Obtaining a representation with such ideal properties is still a formidable challenge.

Concept: Compiler instruction scheduling

An unexploited property of ASCEND III is that each compiled equation belongs to exactly one containing object. Further, the details of the form are determined entirely by the combination of the equation’s symbolic form and the structure of other objects in the same containing object. Thus, compiling equations could 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. We decouple the compilation of equations from the compilation of models in ASCEND IV. This eliminates compiler retry of erroneous equations and saves the user time while new model constructions are being debugged.

Result: Improved modeling productivity

While it is difficult to measure the effects of all the language features described above on the time spent interactively debugging, several users have reported that they are significantly more productive. In one instance, an undergraduate reported creating workable reboiler, condenser, and distillation tray models in three hours by mimicking and extending the style of a basic one feed-two product vapor-liquid flash we provided.

Next we present a more concrete test of our compiler improvements. As noted previously, compilation of equations can be separated into a second pass. In Table 1 we show compiler statistics for the flowsheet shown in Figure 7. This flowsheet contains 17,500 equations. The process separates C3 hydrocarbons: propadiene, propylene, and propane, and it is derived from Abbott (1996). The times here are for first pass model construction only. We describe the second pass equation compilation performance in the second section.

Table 1: Pass one model structure compilation time under SunOS/Sparc 5 110MHz

17,500 equation C3 splitter	No object passing or copy	Object passing	User directed object copy
CPU seconds	68.9	30.1	20.5

The first entry is the time to compile this flowsheet as written in the ASCEND III idiom, a subset of that available in ASCEND IV. The second entry is the time to compile an equivalent flowsheet written in the object-passing, condition-checking, idiom highlighted in this section. The third entry is the time to compile the flowsheet in the ASCEND III idiom using the expert, manually directed copying of prototypical objects as described by Abbott (1996). This entry indicates that the ASCEND IV compiler performance could be further enhanced by incorporating automatic

copying of objects in pass one.

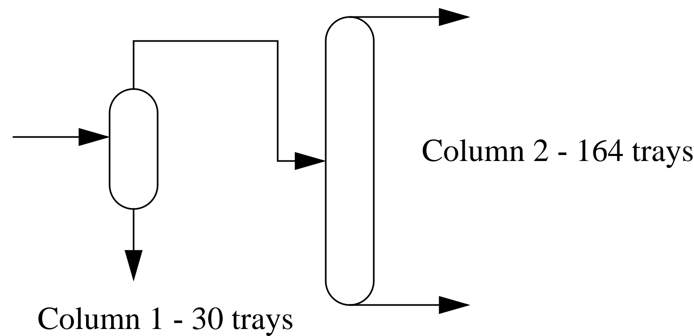

Figure 7 C3s distillation with 194 theoretical trays and 17,500 equations.

Discovering similar objects to improve performance

Virtually all large problems are constructed by repeating small structures which are better understood and adding one or two new features. A 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. We can discover these repeated structures automatically, even though they are anonymous, and use that knowledge of structure to reduce memory and CPU requirements for large models.

We make reductions in compilation time by separating equations into a second compiler pass as described in the previous section. 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 discovered. All similar models can share this copy of the equations. Relation sharing is first recognized by Abbott (1996) in the process of user-directed model copying. A user-directed approach cannot, however, make maximum use of repeated structures and is vulnerable to user misdirection.

Concept: Anonymous type detection

Unfortunately, as we discussed in the introduction, the formal type (class) of an object does not completely determine its internal, hierarchical structure. We must detect the anonymous type (anonymous sub-class) of each object in the model hierarchy in order to identify the repeated model structures. In the user-directed approach, the user is responsible for understanding anonymous type subtleties and identifying important groups of objects sharing like structures.

The results of our first experiment to test the potential impact of this approach are given in Table 2. In this experiment, we use an optimistic algorithm for determining the anonymous type of each object. This algorithm checks the formal type of the object being classified and the already determined anonymous type of each part in the object to determine the object's anonymous type. Once the anonymous types are determined, we compile and share equation structures

as already indicated.

Table 2: Pass two equation compilation time and memory

17,500 equation C3 splitters (194 trays)	No equation sharing	User directed equation sharing	Anonymous type guided equation sharing
CPU seconds	83.1	5.4	3.6
Total space required for completed model (megabytes)	26	14	11

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 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 takes hours on any 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.

There is an interesting fly in our ointment of high performance

Why did we describe the classification algorithm used as *optimistic*? We did so because models compiled with object passing form DAGs and not trees. On a DAG it is not sufficient to check just the anonymous type of each part in an object when classifying the object. Figure 8 illustrates why

1. While the order is the same, the coefficient is somewhat higher to support the flexibility of interactive problem specifications and subproblem management.

this is the case with the simplest possible example.

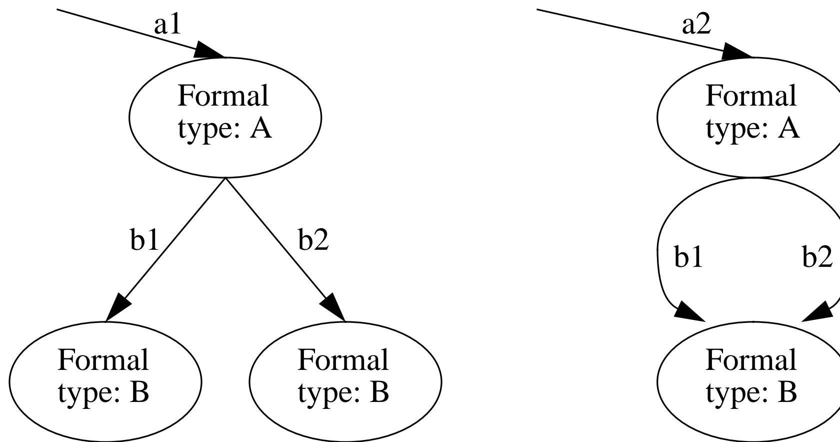

Figure 8 Counterexample instances to an optimistic classification algorithm on DAGs

According to our optimistic algorithm, the anonymous type of objects a1 and a2 is the same. a1 and a2 are both of formal type A and both have to children of anonymous type B. However, a1 and a2 are clearly different data structures with different semantics. Consider an equation in the definition of A, e.g. `b1 = b2`. This equation relates two variables, b1 and b2, in object a1, but merely states a tautology in object a2. Fortunately, this sort of subtlety does not occur in the flow-sheeting models we have tested. Indeed, in the example of Figure 2 and Figure 3 this structure is precluded by the WHERE conditions which state that the stream parts of the flash model will not be the same object.

Imagine, however, a user trying to find the compiler's optimization error with a similar structure buried in a model of 100,000 equations. A compiler must be correct all the time, not just most of the time. We are presently investigating more rigorous algorithms to detect anonymous types in the presence of such subtleties. Any mathematical programming system which passes pointers in the construction of its object data structures needs to recognize the subtle effects of anonymous type within DAG structures in order to use the equation sharing required for scale-up to large models.

Conclusions

In the first part of this paper, we presented new modeling language formalisms which increase the probability of successful reuse and modification of library models for large applications. We observe that the same language features which make model reuse easier for the modeler also make compiler performance substantially better. We note the use of DAG model structures to obtain scalable modeling systems.

In the second section of this paper we introduced the new issue of anonymous type detection in DAG structures. We suggest the ways in which anonymous type detection can transform today's modeling systems with large object-oriented overhead memory and CPU costs suitable only for small model projects into scalable systems capable of performing on the same level as hand-coded binary library based flowsheeting systems. Based on promising early results presented here, we will continue to investigate the anonymous type detection issue. 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.

References

Abbott, K. A. *Very Large Scale Modeling*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA USA (1996).

Allan, B. A. *A More Reusable Modeling System*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA USA (1997).

Allan, B. A., V. Rico-Ramirez, M. Thomas, K. Tyner, and A. Westerberg. ASCEND IV: A portable mathematical modeling environment. ICES technical report, number not yet assigned (1997). available via http://www.cs.cmu.edu/~ascend/ascend-help-BOOK-21.pdf.

Andersson, M. *OMOLA - An Object-Oriented Modelling Language*. PhD thesis, Lund Institute of Technology, Department of Automatic Control, Lund, Sweden (1990).

Barton, P. I. *The modeling and simulation of combined discrete/continuous processes*. PhD thesis, Department of Chemical Engineering, Imperial College of Science, Technology and Medicine, London (1992).

Marquardt, W. An object-oriented representation of structured process models. *Computers and Chemical Engineering*, 16S:S329–S336 (1992).

Meyer, B. Applying 'design by contract'. *IEEE Computer*, pages 40–51 (1992a).

Meyer, B. *Eiffel: The Language*. Prentice Hall (1992b).

Piela, P., T. Epperly, K. Westerberg, and A. Westerberg. An object-oriented computer environment for modeling and analysis: The modeling language. *Computers and Chemical Engineering*, 15(1):53–72 (1991).

Piela, P. C. *ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania (1989).

Rico-Ramirez, V., B. Allan, and A. Westerberg. Conditional modeling in ASCEND IV. Technical Report EDRC/ICES TR number not yet assigned, ICES/Engineering Design Research Center, Carnegie Mellon University (1997).

Westerberg, A. W., K. A. Abbott, and B. A. Allan. Plans for ASCEND IV: Our next generation equational-based modeling environment. Boston, Massachusetts. ApsenWorld '94 (1994).

Westerberg, A. W. and D. R. Benjamin. Thoughts on a future equation-oriented flowsheeting system. *Computers and Chemical Engineering*, 9(5):517–526 (1985).

Zaher, J. J. *Conditional Modeling*. PhD thesis, Department of Chemical Engineering, Carnegie Mellon University (1995).

Piela, 1989

Allan, 1997

Rico-Ramirez et al., 1997

Abbott, 1996

Zaher, 1995

Westerberg and Benjamin, 1985

Allan et al., 1997

Meyer, 1992b

Meyer, 1992a

Piela et al., 1991

Westerberg et al., 1994

Marquardt, 1992

Andersson, 1990

Barton, 1992