Contact: Anthony J H Simons

Address: Department of Computer Science, Regent Court,
University ofSheffield, 211 Portobello Street, SHEFFIELD S1 4DP.

Telephone: (UK) 742 768555 ext 5572
(UK) 742 825572 (direct line)

Facsimile: (UK) 742 780972

Email: A.Simons@dcs.shef.ac.uk

An Optimising Delivery System for Object-Oriented Software

Anthony J H Simons, Low Eng Kwang and Ng Yee Mei, Department of Computer Science, Regent Court, University of Sheffield, 211 Portobello Street, SHEFFIELD S1 4DP.

Abstract

An open-ended and flexible object-oriented language ideally requires its compiler to optimise code at the time of building complete application systems. Optimum code can be judged in terms of its size, which should be compact, and speed of execution, which should be fast. In addition, the turn-around time for recompilation should be as short as possible. We have developed four strategies, based on graph analysis, which allow these objectives to be attained in a more consistent manner than with current compilers. Our development system traces inter-class dependencies automatically to minimise the number of passes over source files at compilation. Our optimiser then performs three tasks: firstly it transforms the system of classes ported into the target application by collapsing inheritance relationships; secondly it determines whether methods should be statically or dynamically bound for that application; and thirdly it removes the unnecessary levels of nesting in call graphs which arise from strict encapsulation, by detecting safe cases for inlining which are also guaranteed to reduce code size. Our findings are being incorporated in BRUNEL, a new object-oriented language and programming environment, which aims to provide a high degree of support for abstract design, while generating optimal code for applications.

1. Introducing the BRUNEL Project

For the last three years, a design team at the University of Sheffield has been investigating how to increase the level of run-time efficiency in object-oriented systems, without sacrificing open-ended flexibility or security. The main findings are being incorporated in BRUNEL, a next-generation object-oriented language and programming environment aimed at the software engineering industry [Sim91]. The language is named after one of Britain's most famous and far-sighted engineers, many of whose engineering projects from the Victorian era still fulfil their function today.

BRUNEL is designed to overcome perceived limitations in the correctness, efficiency and transparency of current object-oriented languages and development systems. The BRUNEL concept rests on three main pillars:

- The type model for BRUNEL is elaborate and derives from Category Theory, following the tradition of [CW85, CCH89a, CCH89b]. In this model, classes are considered to be polymorphic abstract types, equivalent to parameterised type constructors [SC92, Sim93], whose type parameters may be replaced at compile-time or run-time. The polymorphic abstract types are represented using parameterised sets of function signatures and axioms. The language is strongly and statically typed, with a complete polymorphic type checker which ensures signature and assignment conformity at compile-time and axiom verification at run-time.

- The compilation model for BRUNEL rests on the principle that a specific delivery system will typically require many fewer classes and inter-class dependencies than those expressed in the developer's class libraries. It requires a very high standard of system support for monitoring inter-class dependencies, incorporating an automatic loading and configuration tool, a class graph structure optimiser, an automatic procedure for determining static or dynamic binding and an optimisation process for replacing unnecessary nested method calls by inline expressions. A further requirement is that BRUNEL compilers for different platforms should produce semantically equivalent systems on each platform, including semantically equivalent representations of simple types such as reals and integers.

- The development environment for BRUNEL provides a graphical interface to class systems and subsystems in the tradition of the best object-oriented browsing environments [Gol85, BS83]. Unlike many CASE tools which are restricted to the generation of design documents and code stubs, the BRUNEL environment aims to have a full reverse-engineering capability, generating abstract views and interactive diagrams from source. The graphical user interface will conform to industry standards of the day for presentation and portability. The current version [Ong91, Tam92] is implemented in OpenWindows following the recent trend in standardising window, icon, menu and pointer (WIMP) interfaces.

Here we describe the progress achieved so far mainly in the second pillar, which we consider to be one of the more novel aspects of the BRUNEL project. The optimising techniques we report will be of a wider interest to designers of development systems and compilers for other strongly-typed object-oriented languages, such as Eiffel, Trellis and C++. In many cases, similar advantages may be obtained for these languages. However, the design of the BRUNEL language and its close coupling with the development environment bring certain benefits which may not be realised fully in other languages.

2. Some Issues in Compiling Object-Oriented Languages

Object-oriented languages present significantly greater challenges to compiler-writers than other structured languages. Developing a compiler which will generate a run-time system with precisely the same semantics as that expressed in the design of the class hierarchy may take up to four or five times the effort invested in developing a standard one-pass compiler for a block-structured language like Pascal [How93]. Indeed, the designers of Eiffel

developed their first compiler from a solution to what they had initially supposed were an insoluble set of constraint equations [Mey88].

A conventional compiler for Pascal performs the familiar four phases of tokenisation, syntactic parsing, semantic analysis and code generation. During code analysis, a compiler assembles symbol tables storing the names and types of the symbolic identifiers used in the program. Typed variables, record structures and procedure headers are retained for the duration of their scope, such that expressions which later use them can be verified for syntactic and semantic correctness. Parse trees for each procedure body are constructed; these need only be retained until code for that procedure is generated. Since the syntax rules of Pascal ensure that simple declarations are always ordered before more complex, dependent declarations in a file, Pascal programs can be processed efficiently in a single pass. A compiler may choose to perform each phase in sequence globally for the whole program or, where memory restrictions prevent the retention of the whole program, it may pipeline them, analysing and generating code in completed chunks. Below, we describe ways in which object-oriented languages defeat this simple model of compilation, introducing the additional tasks faced by compiler writers.

Because of the highly modular nature of object-oriented software, the classes required for any one application may be obtained from different source files (and directories). Units of compilation vary widely in current languages. Some, like C++, CLOS or Object Pascal, practise a style whereby small groups of related classes are usually placed in a single file. Other languages, like Smalltalk or Eiffel, insist that each class be maintained separately. Clearly, the latter approach is more flexible since it allows classes to be added or deleted singly in applications. Either way, this presents a practical problem in tracing inter-class dependencies at compile time: firstly, can a logical order be found for processing files and secondly, can an optimal order be found for opening files?

The first problem concerns whether it is in fact possible to guarantee the correctness of an object-oriented program. A class can be viewed approximately as a partial record structure whose components are attributes and methods (ie data storage and functions owned by the class). A class depends both on its ancestors, from which it may inherit its remaining attributes and methods; and on any supplier-classes whose data it uses or whose methods it calls. Logically, this prior information must be processed before the class itself. However, there exists the possibility of circular dependencies, which must safely be broken before the class can be processed. In section 3 below, we describe in more detail the nature of inter-class dependencies and an algorithm which can be used to break dependency cycles.

The second problem concerns the fact that, due to the complex nature of inter-class dependencies, it is unlikely that source files can be processed in a single pass. When compiling a complete object-oriented program from source, the aim should be to open each source file as few times as possible, while constructing the dependency information needed for syntactic and semantic checking. Language environments which support the automatic detection of inter-class dependencies, such as Eiffel's, require up to four passes of each source file, in which the local syntax is checked, inherited material is incorporated, routine calls dependent on other classes are checked and code is finally generated. This can lead to a significant time overhead on some filing systems. In section 3 below we describe how we have reduced this requirement to a maximum of two passes.

Of course, it is more customary to assemble large applications from collections of source- and object-code files. In this model, individual classes (or program modules containing a small group of related classes) are compiled to object-code, supplemented by a header file containing data declarations and method type signatures. When the application is assembled, only the remaining new code elements need be analysed in full; previously compiled code elements are simply linked in the final assembly stage, using the header files to guide syntactic and semantic checking. This mode of working reduces locally the turnaround-time for recompilation, since only those header files on which a class (or program) is immediately dependent need be consulted. However, the total number of passes over the source code during the creation of an application may have been the same. More importantly, incremental compilation of this kind tends to fix the state of program modules too early in the development of applications. It introduces the need for a dependency monitoring system, whereby all dependent object-code modules invalidated by a late modification to a class are re-compiled. Eiffel systems will typically manage this automatically (with the attendant multiple passes through source), whereas C++ systems usually leave this task to the programmer - either on the command line, or through maintenance of a "makefile" in which dependencies are recorded (with the attendant scope for human errors).

One element crucial to minimising the size of executable systems is to have a smart linker. Unfortunately, some compilers, upon encountering an *#include* directive, will link the entire object-code module. Instead, we want to link only those methods which are actually used by the client class (or program). This is described in section 5 as part of a general binding and linking strategy. Inheritance introduces further complexity here. Often, all object-code modules for ancestor classes are *#included* indiscriminately in the final program, complete with their structural templates. However, we only need those extra structural templates if, at some point in the program, we want to treat an instance of a class as though it were an instance of one of its ancestors. If not, we can simply construct a single structural template for a terminal class in the inheritance graph (ie one which will be instantiated in the application) from information obtained from many places in the inheritance graph. Although it is desirable to maintain classes at all points in the graph where significant abstraction may be captured, it is not clear, until the application is linked and built, how many of the intermediate classes require a separate representation in the application. A compiler should attempt to minimise the inclusion of such intermediate structures. In section 4, we describe a novel series of transformations on the subset of classes required for a given application which collapse the inheritance relationships between them, pruning out intermediate classes.

The inheritance of methods presents its own challenges. When compiling a method defined on a particular structural template, a compiler uses this information to calculate the offsets of attributes in objects of that structural type. Descendent classes will have a different structural template, due to the addition of attributes, which means that applying an inherited method to an object could result in the wrong offsets being accessed and updated - the object's data could become corrupted. Fortuitously, in single inheritance schemes, the structural template of a child class simply appends data storage blocks monotonically to the parent's template. Methods accessing the parent's attributes will automatically access the same, correct offsets in the child. On the other hand, multiple inheritance schemes present a difficult problem, whereby the offsets of attributes in a child class may be displaced with respect to their declared positions in the parent class. If the inherited templates of multiple parents are concatenated in order, any class inherited out of direct line (ie a second, third, ... parent) will suffer from displacement (see also Figure 5 in section 4). Many schemes for overcoming this

difficulty have been proposed: these include recompiling methods inherited out of direct line, using extra levels of indirection to access sub-parts of classes and pointer arithmetic. We have reviewed the main approaches in [Ng92] and eventually settled for a scheme similar to [Str91] which uses pointer arithmetic and inherited class templates. We selected this mainly because it reduces dramatically the number of recompiled versions of the same method in target applications, without significantly compromising access times into structures. The interested reader should consult these sources for further details.

Inheritance brings with it the polymorphic application of methods, in the strict sense that they apply directly to objects of more than one type. Object-oriented languages extend this concept to polymorphic abstract methods, whose algorithms are implemented in a family of methods, each one associated with a different type of object. In section 5 we discuss how this gives rise to dynamic binding, whereby one or other variant of a method is selected at run-time by discriminating on the type of the object. To do this, a compiler has to construct what is known as a *dispatch table* for some methods in an application. Instead of placing a direct call to the compiled method in the object-code, the compiler places an indirect call to a method obtained from the dispatch table, accessed by some index computed from the type of the object and the name of the method. A dispatch table may be very large, growing in proportion to the product of classes and methods that use dynamic binding. The overhead of an indirect call adds to system execution time. It is therefore important to tailor the amount of dynamic binding used to the needs of the application.

We have summarised the options for minimising the size of dispatch tables and optimising the speed of the indexing system in [Ng92]. For some languages, such as Smalltalk and Objective C, this is of paramount concern, since these languages have a strong commitment to dynamic binding. For other languages, such as C++, the size of the table is kept small and the indexing system correspondingly simple due to the fact that the programmer has to flag dynamically bound methods explicitly in the source code. In section 5 we discuss how this compromises flexibility and the freedom to extend class hierarchies. Instead, we adopt an automatic approach to the detection of static and dynamic binding. Although every method should be written as though it could be selected dynamically at run-time, a compiler should be able to determine, at the time of building the application, a large set of calls for which only one method can be invoked and replace the dynamic lookup by a static method call, or even by an inline expression. Our algorithm performs a global optimisation for a given application and relies to a certain extent on BRUNEL's novel type system. A slightly less optimal solution can be found for other strongly-typed languages.

Object-oriented languages typically promote encapsulation, or the hiding and protection of a class's internal data. This gives rise to an increase in interface methods, whose sole purpose is to give controlled access to a class's internals. In section 6, we describe cases in which this escalates to an intolerable degree. A natural solution is to request inline expansion for these, and other, short methods. This brings an immediate speed advantage. However, inlining is not always without hazards - it prevents redefinition of the inlined method and also may give rise to the multiple evaluation of sub-expressions. Furthermore, an aggressive inlining strategy stands to increase the size of the application code. We suggest a scheme for detecting automatically safe cases when interface methods can be inlined, with a guaranteed reduction in code size.

3. Tracing Inter-Class Dependencies

To achieve the fastest turn-around at compile time, the set of classes required in a delivery system should be processed in order of dependency. However, the circular relationships in object-oriented software usually preclude any simple approach to compilation. We have established a constraint-based approach [Sim91, Low91] to determine how cycles of mutual dependency may safely be broken.

First, let us consider the constraints. A class may depend directly on another class in several different ways [Mey88]. It may *inherit from*, or be a *client of* another class. Then, through the transitive nature of dependency, a class may depend indirectly on others.

The inheritance relationship is important, since the full description of a class's structural and behavioural type can only be obtained by fetching down all attribute (data storage) and method (function) declarations from its ancestors. Inheritance satisfies the properties of a partial order, in that it is a reflexive, transitive, antisymmetric relationship. Inheritance graphs are directed and acyclic, with the result that any set of dependencies may always be resolved to chains and a total order may be determined for processing these chains. Inheritance is equivalent to a containment dependency, which we define below.

The client relationship is more complex, since a class may use the facilities of another in several different ways. Classes frequently describe structures composed out of simpler elements - this is one kind of client dependency. Composition may involve the direct inclusion of the simpler type, which we describe as *containment*, or else an indirect *reference* to the simpler type. In the first case, the structural template of the simpler element is expanded inline in the template of the containing class; in the second case, a standard pointer to the simpler type is stored instead. Classes may enter into recursive, mutually recursive and cyclic referential dependencies, but cyclic containment is prohibited, since otherwise a structure might have infinite size.

Classes also frequently collaborate with others through their methods - this is another kind of client dependency. Collaboration may involve having another class as an argument type or return type in a method (here, we could also include local variables). A method may pass its arguments *by value* or *by reference*. In the first case, the structural template of that argument is needed to compile the method. In the second case, space for a standard pointer is required. In both cases, the type signatures of the methods offered by the collaborator are required, since the main purpose of collaborating is to invoke one or more of the collaborator's methods and we should need to check that such calls are correctly made. It is easy to see that cyclic dependencies arise through mutual collaborations between classes. Fortunately, a class's structural template can be constructed without reference to its methods.

From these considerations, we assemble the following binary constraints on the order of processing classes and methods:

a) all unprocessed classes P referenced inside other classes Q should generate typed pointers which are processed before Q;

b) all classes X contained inside other classes Y, whether through inheritance or through composition, should be processed before Y;

c) all arguments A required for method bodies B, whether passed by value or by reference, should be processed before B;

d) all type signatures S required to check calls in method bodies T should be processed before T.

The only additional compositional rule is that:

e) cyclic containment is prohibited.

By exploring the interaction of these constraints, we see that (a) and (c) require pointers and structures to be generated before method bodies are processed. Similarly, (d) when applied recursively requires type signatures to be processed before method bodies. The compositional rule (e) ensures that a class may not inherit from its own descendant, a class may not be ultimately part of its own component and a class may not be ultimately part of its own descendant.

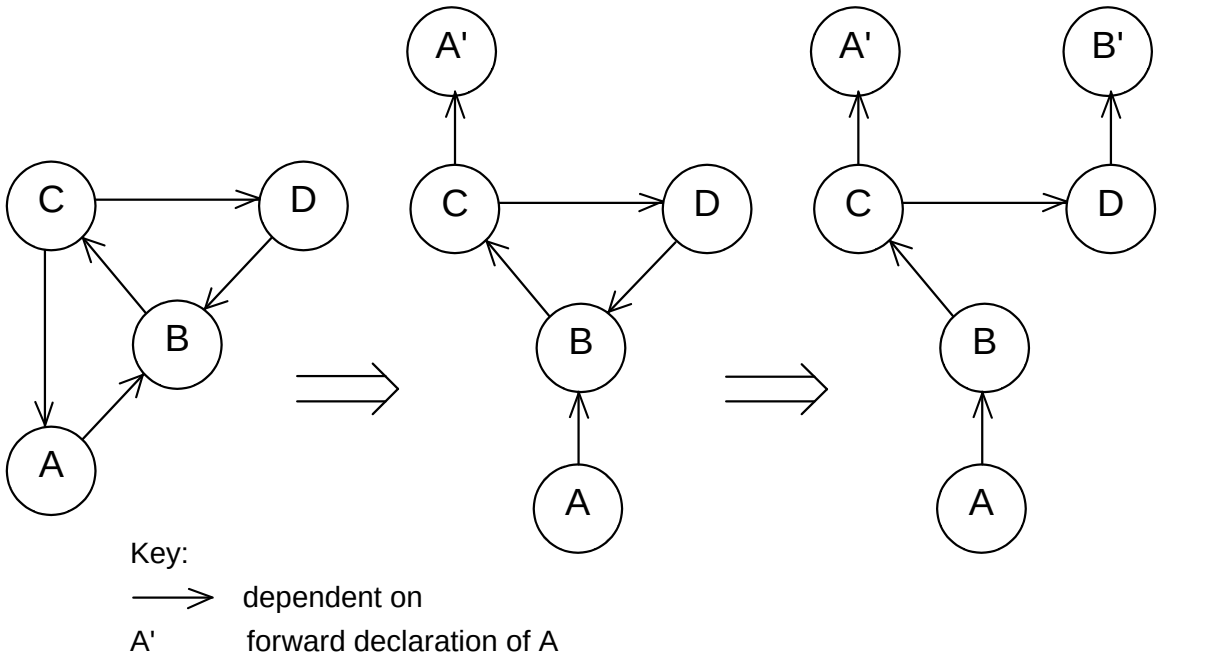

Figure 1: Breaking complicated dependency cycles

Our model for compilation treats the task of resolving these constraints as one of reordering the textual information in the class source files as though it were to be presented to a single-pass, recursive descent compiler. The prototype version of BRUNEL does in fact perform a cross-translation into ANSII C, which is used as a portable pseudo-assembler. A correct, but usually sub-optimal solution to these constraints is to:

generate a typed pointer for every class used in the application;
generate a structure template for every class in order of containment;
generate the type signature for every method used in the application;
generate the object code for every method body.

There is therefore (at least) a logical order in which class data can be processed, although the prospect for minimising the number of compiler passes does not yet look promising! This accommodates the worst case, where global cyclic dependencies permeate the application. Typically, we can reduce the number of forward declarations of typed pointers and type signatures that have to be generated, using a simple graph-searching technique to establish pre-order.

Figure 1, adapted from [Low91], illustrates the basic strategy for creating forward declarations of type signatures. Classes A - D have a client relationship with each other through invoking each other's methods. Assuming that the search starts from class A, dependencies are traced recursively until the whole graph has been explored (either breadth-first or depth-first). Any class encountered for a second or subsequent time requires a forward declaration of its methods' type signatures. The interim stage shows the point at which class C is being explored; a forward declaration A' is requested. The final stage shows the point at which the entire graph has been explored; a forward declaration B' has also been requested. Now, provided that type signatures for A' and B' are generated first, the methods for the classes shown can be parsed and compiled in reverse order of dependency, namely D, C, B, A, avoiding the need to declare forward C' or D'.

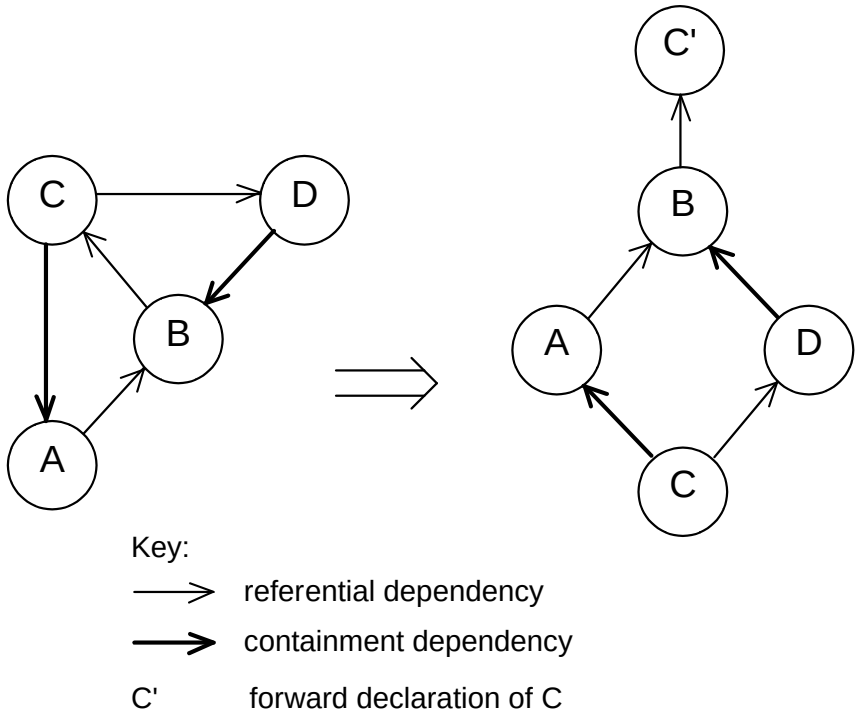

Figure 2: Preferential resolution of containment

We could, of course, have made the nodes in the graph individual methods, rather than whole classes. Exploring a call-graph in this way would allow a much more accurate assessment of the need for signature declarations. However, such a strategy would then require class source files to be opened on a piecemeal basis to parse each method, something which we are trying to avoid. Secondly, we link this analysis with the algorithm for determining the order of containment and detecting the need for forward declaration of typed pointers.

The astute reader will notice that there are a number of solutions to the problem above; and that the one illustrated does not, in fact, generate the minimum number of forward declarations. A less simple-minded algorithm takes into account the degree or weight of dependency. Class C is doubly dependent and so should be allowed to migrate to the head of the dependency cycles CAB and CDB.

This is the case in Figure 2, which also illustrates how the search algorithm defers resolving referential dependency in favour of containment dependency. Here, classes A - D have referential dependencies as illustrated; but the more important containment dependencies are highlighted in bold: C inherits from A and D contains an attribute of type B. Starting from class A, the algorithm proceeds via B until it encounters class C. For two reasons, C must migrate to the head of the dependency cycle CAB: it contains A and it depends on more classes than any other so far. The cycle is broken and a forward declaration C' is requested for the sake of B. Then, from C the algorithm proceeds to D which contains B. Here, the algorithm would have detected a second dependency cycle DBC, except that the cycle has already been broken. D is left as the head of the chain DBC'; its weight is greater than any other by virtue of the containment of B. At this point, the graph is fully explored and all cycles have been broken, leaving two solutions for compilation. Now, provided that a forward pointer C' is declared, the classes may be compiled in reverse order of dependency, either BADC or BDAC, avoiding the need to declare forward the pointers A', B' and D'.

The order of search for these two algorithms is the same; although they may construct different dependency chains for class structure and method body compilation. They produce identical dependency chains in many cases. This is because a class containing simple types (such as integers) tends to provide methods passing these by value; whereas a class referring to complex structures (such as editor documents) tends to provide methods passing these by reference. The advantage of having the two coincide is that all of a class can then be compiled in a single pass. The remaining problem consists in making enough dependency information available in the program development environment so as to minimise the number of times source files are re-consulted during compilation.

```
$N INT_COORD
$B INT_COORD[C2]
...
$C OBJECT INTEGER
$M INTEGER C2 BOOLEAN
...
$S x : INTEGER
$S y : INTEGER
$S moveTo INTEGER INTEGER : C2
$S equal C2 : BOOLEAN
...
```

Figure 3: Extract from Header Section

In the current BRUNEL prototype system, classes are stored in individual source files in a semi-processed format which is invisible to the programmer. The visual presentation of the language is controlled entirely by the development environment, allowing certain indexing and type signature information to remain hidden. Classes are typically created using dialog

boxes and a form-filling approach. The hidden source files contain a header section and a main section. Dependency information is encoded in the header section in a format which is readily retrieved and updated automatically by the BRUNEL system development environment [Tse91], about which more below. Figure 3 shows an extract from the header section for a simple integer coordinate class, called INT_COORD. The class inherits from OBJECT and contains two INTEGERs. It has methods in the types INTEGER, BOOLEAN and in its own type, which is represented by a parameter C2. Dependency information is encoded in lines $C, $R and $M, representing containment, referential and method dependencies - this class happens to have no $R referential dependencies. In the Figure, the $N line simply names the class, the $B line expresses the bound on the type parameter C2, which we shall elaborate in the next section and the $S lines describe the type signatures of all the methods known by this class - including inherited methods, whose type signatures may have been modified as a result of inheritance (see section 4).

Note that the header section is not exactly equivalent to an interface section [CN91] or definition module [Wir82], since it contains other secret information needed to resolve inter-class dependencies. For example, it also contains parent and child dependency links to allow propagation of edit changes in the inheritance hierarchy. In any case [Mey88] criticises so-called interface descriptions in Ada for their inclusion of information specifying the secret representation of a type, though this is clearly needed for single-pass compilation. For this reason, we do not consider header information as an interface; when such an abstract view is required, the development environment filters the header section, presenting only the type signature aspects of an interface to the programmer.

The BRUNEL system development environment maintains an internal model of dependencies in the the class graph and this model is constructed on the fly, as classes are touched during a programming session. We prefer this approach, since it brings forward some of the tasks involved in compilation to the edit session itself. The compile-time model of the class dependency graph is constructed incrementally in a way which is less noticeable to the programmer. In the worst case, this degrades to a two-phase strategy, 'peeking' at the header portion of class files to construct the dependency graph, then a main pass which presents the rest of the class files to the compiler in an optimal order. Alternative approaches include keeping separate header- and main-files [Str91] (with double the overhead for the underlying filing system and no clear speed advantage when compiling from source), or maintaining a single system image file [Gol85] which provides indices into all the source files (with associated time penalties on first-time access and for a controlled exit). The BRUNEL approach lies between these two, building its model of system dependencies on a need-to-know basis.

For the moment, we do not compile incrementally to object-code. This is partly because we wish to generate portable C applications, but more significantly because we make global system-wide optimisations which would not be possible with early generation of object-code. Instead, we opt for an early analysis of source. The global optimisations we describe in sections 4, 5 and 6 may eventually give rise to a new generation of object-code compilers and linkers, at which point it would make sense to build a native code compiler for BRUNEL. A refinement to our current approach might be to store complete parse-trees for the main sources in each file, instead of semi-processed text. Recent developments with the Mjølner BETA language [Mad93] and Mjølner ORM environment [MDB90] for Simula suggest that this is a tractable proposition with further benefits in the form of syntax-directed editing.

Whereas in our form-filling approach the semantic and dependency implications of expressions are checked on entry to the environment, full syntax-directed editors are more flexible, since they can be reconfigured to accept a different BNF. The storing of parse-trees would remove another analysis stage from compilation, but would require further effort in visualising the textual form of the code to the programmer. Also, syntax-directed editors do not accept incomplete textual forms; this would prevent the acceptance of program jottings or mal-formed code stubs.

4. Collapsing the Inheritance Graph

Inheritance is expressive as a means of sharing type and implementation. BRUNEL supports strict inheritance (of type) and, dependent on this, some lax inheritance (of implementation).

```
class OBJECT[O] : TOP[T]
{  type
     identity () : OBJECT[O];
     equal (OBJECT[O]) : BOOLEAN;
   ...
}

class INT_COORD[C] : OBJECT[O]
{  data
     x, y : INTEGER;
   type
     x () : INTEGER;
     y () : INTEGER;
     moveTo(INTEGER, INTEGER) : INT_COORD[C];
   ...
}
```

Figure 4: Schema for inheritance in BRUNEL

In recent years, much has been made of the independence of *class* and *type* in languages like POOL-I and CommonObjects [Ame90, Sny87]. Here, classes are often viewed simply as units of convenience for bequeathing implementation details to their descendants. A type hierarchy is maintained separately from the class hierarchy to determine subtyping relationships (sometimes, optimal type- and class-inheritance structures can be shown to link nodes in completely different orders). In contrast to this, BRUNEL has a single syntactic construct, the class, which has three related semantic interpretations: *class*, *type* and *template*:

- A *class* in BRUNEL is a higher-order type, or type constructor [SC92, Sim93] whose type parameter(s) are recursively instantiated in descendent classes to derive new type-bounds for inherited methods.

- A *type* in BRUNEL is generated by replacing all remaining type parameters in a structure. Since type-sharing is viewed in terms of behavioural compatibility, the sharing of (implicitly retyped) methods and axioms is therefore possible and supported.

- A *template* in BRUNEL corresponds to the concrete implementation of a type, specifically the table of attributes defined for each class.

The classes of POOL-I and CommonObjects therefore correspond most closely to implementation templates in BRUNEL, whereas classes in BRUNEL are higher-order types, or polymorphic constructions containing type parameters.

Inheritance in BRUNEL has the effect of restricting the set of types that can belong to a class. It does so by changing the type-bounds on type parameters obtained from the parent class, by adding functions to the child class and by progressively implementing the class in a concrete way, as illustrated in Figure 4. Here, the class OBJECT[O] inherits from the vacuous and all-embracing class TOP[T], the top of the inheritance graph. OBJECT[O] provides methods for identity and equality in its own type, represented by the parameter O. This parameter unifies with the parameter T in TOP[T]. The class INT_COORD[C] inherits in turn from OBJECT[O] and provides some concrete implementation (the *data* section) and further functions. The complete set of functions for INT_COORD[C] is: *identity*, *equal*, *x*, *y*, and *moveTo*. However, as a result of the unification of C with O, *identity* and *equal* have become functions in the new self-type C. The constraint on types satisfying C is INT_COORD[], a more severe restriction than OBJECT[]. The child class can be regarded as a more specific type generator. Note that we are not in the domain of types, so we are not asserting that INT_COORD is a subtype of OBJECT. The process of type parameter instantiation and type-checking is briefly introduced in section 5.

A syntactic class definition in BRUNEL may therefore provide anything from a completely abstract specification (cf a fully deferred class in Eiffel) in terms of method stubs whose properties are verified by axioms (not illustrated here), to a concrete specification, detailing attribute storage and complete implementations of methods. Descending an inheritance graph in BRUNEL typically describes a process of type restriction and may eventually describe a process of reification (introducing a particular concrete representation is just one kind of restriction). Concrete attributes introduced at points in this graph must always be pertinent to the class and all its descendants. BRUNEL also allows the marking of certain attributes as shared by all instances of a class (cf class variables in Smalltalk).

Inheritance rightly encourages the creation of a great number of polymorphic classes, many of which are only incrementally different from other classes. A multiple inheritance graph may be seen as mapping out focal points in a space of overlapping type descriptions [SC92]. However, generating a type and a template for each node in the hierarchy presents a problem from the space-efficiency viewpoint. Application programs will, in general, only need to generate types and templates from specific terminal class descriptions in the network. These classes will have inherited most of their specification and implementation from intermediate classes in the network. Many of these intermediate classes will not require a separate representation in the target code, since their only purpose is to bequeath inherited material.

Our commercial and industrial contacts have reported how programmers, having been trained to factor out the functionality of a system over some set of classes in an inheritance graph, are then surprised to find that applications are too large to load onto standard processors [Qui90]. A particular set of bad experiences was obtained when converting from the older single-inheritance version of C++ to the multiple-inheritance version [Str91] and finding that compiled applications would no longer fit on 80386 processors, despite the apparent reduction in source text.

The problem appears to have been due to the overhead in maintaining many finely-factored class templates in the runtime system. One solution is to try to increase the grain size of objects. This leads to a style where there are fewer objects, which are more multi-functional. The pressure is on the programmer to produce, every time, a class which maximises operational code at the expense of conforming to type. One example of the undesirable effect of this pressure is Eiffel's POLYGON class in [Mey88] which, in addition to being the abstract ancestor of all SQUARE, RECTANGLE, ... classes, is also used as the concrete class to create N-vertex polygons. As a result, its routine *add_vertex* cannot be exported in its descendants (squares clearly cannot add to their vertices) - this is formally equivalent to deleting a function from the type's interface, thereby invalidating the type-status of the inheritance graph.

BRUNEL observes a strict type discipline; therefore we should expect to see more classes and more finely factored classes in BRUNEL than in some other languages which permit arbitrary implementation sharing. In consequence, we require the compiler to reduce automatically the number of class-generated types and templates represented in the target language to the essential minimum for particular applications. This process of pruning and collapsing the class graph is described as an ordered sequence of transformations:

a) The compiler is invoked on the class which encapsulates the whole application. Dependencies are traced recursively from this to all other classes in the application (see section 3). A dependency is either an *inherits from* or *client of* relation. The former produces a transitive closure of *ancestor* classes, whereas the latter produces a transitive closure of *supplier* classes.

b) A subgraph of the application system is then constructed. This graph contains the application class and all its (recursive) ancestors and suppliers.

c) Fully abstract classes whose sole purpose is to provide specifications (ie with completely deferred implementations) are eliminated after static type checking, but their axioms are maintained for inclusion in descendent classes. Axioms may be data type invariants, or pre- and post-conditions to methods. However, partially deferred classes which are the branch-points for dynamic dispatching are treated in (e) below.

d) Intermediate classes, which are not instantiated in the application, but which share some physical structure (such as shared attributes) among terminal classes in more than one descendent branch, are reduced to a shared data structure, or table. References to shared attributes, once their scope has been checked, are compiled out to offsets into this table.

e) Intermediate classes, which are not instantiated in the application, but which share operational code (ie implemented, not deferred methods) among terminal classes in more than one descendent branch, generate a single template and one set of methods. The template is used to calculate the offsets of attributes accessed by the methods. BRUNEL employs a multiple inheritance scheme very similar to C++ v2.0 [Str91] in which such templates are applied to different offsets in objects prior to accessing attributes.

f) Terminal classes and any intermediate classes which are instantiated in the application must be fully represented in the target language. These both generate a template, a type schema and one (possibly empty) set of methods. The template is used as above and the type

schema is used during the automatic detection of static and dynamic binding (see section 5 below).

g) All other intermediate classes are "flattened" down to the nearest class represented in the application. "Flattening" is the process whereby a chain of classes inheriting from each other is collapsed down to a single datatype which declares in one place all the inherited features.

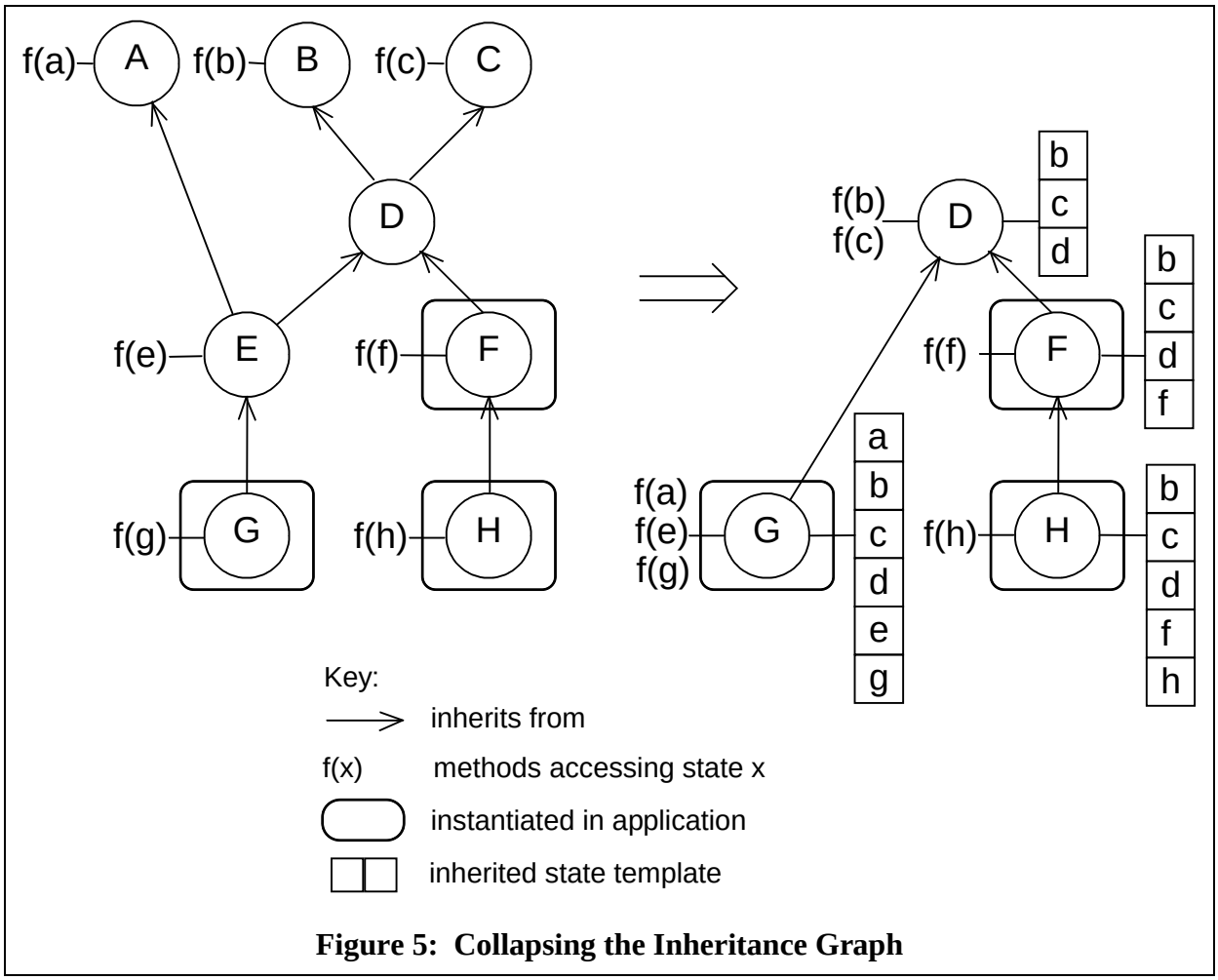

Figure 5: Collapsing the Inheritance Graph

Figure 5 illustrates the transformation of an example class graph. Each class A - H in the class library is assumed to provide a set of methods f(a) - f(h) and a block of attribute declarations a - h. Only classes F, G and H are instantiated in the application, so these must appear in the collapsed graph. Class D collects inherited material from B and C which have been "flattened"; however D must generate a template so that the single copies of methods in f(b), f(c) access the correct offsets in instances of F, G and H. The importance of this template is seen especially in G where the attributes collected in D are inherited out of direct line, due to E inheriting multiply from A and D. D's template, incorporating the blocks of attributes b, c and d, is applied to an offset in G before any method in f(b), f(c) is called on an instance of G. Classes A, B, C and E are eliminated in the collapsed graph.

We have had some success in reducing the size of simulated run-time systems using this approach. From the small-scale modelling tests [Low91] we have carried out so far (systems

of 10 - 100 classes) on a simulation of the optimiser, it appears that we can obtain a reduction of between 32% and 50% on the number of types and templates generated at compile time. Counting each instance variable declaration as a uniform cost, this represents a total reduction of between 20% and 38% in data declarations for the application system. All the test examples were systems of finely-factored classes using multiple inheritance. The classes instantiated in the application were situated between depth 3 and 5 in the inheritance graph. The improvement was calculated with respect to the subgraph extracted from the class library and not with respect to the original library. It would be useful to have some form of statistical indication of how effective this technique is over large, actual production systems; however we do not yet have enough data to quantify this.

This algorithm can be viewed as a way of automatically extracting larger, more multi-functional classes from a finely-factored class library. It gives the developer the freedom to engage in abstract design, without the burden of implementation considerations. The best results are obtained where the developer's class library is quite large, with many intermediate nodes corresponding to branch points for other class variants not used in the current application. These nodes would have a separate representation in Eiffel and C++ systems, but are collapsed into their descendants in our system. Our technique could be incorporated into a C++ to C translator, but not into current C++ native code compilers. This is due to the early fixing of classes in object-code. Methods compiled over a C++ base class automatically need that class's template, even if the application only uses direct-line descendants of that class. Our technique might be incorporated into Eiffel in the following way. The Eiffel source for a class is typically translated to C and compiled to one or more object-code files. These, and other files which record dependency information, are hidden in a <name>.E directory, corresponding to the <name>.e source file. Eiffel is able to check whether dependent files need recompilation. This approach could be adapted to generate alternative collapsed C sources and object-code files for classes used in particular applications projects. The idea is that Eiffel could check for changes in different subsystems of classes being developed for a particular application. If the inheritance structure of the subsystem changed, then a different set of intermediate C sources would be generated. While developing a given application, this would limit the number of re-generations of C sources and compilations to object-code. When switching to a new project, system-wide recompilation would occur.

5. Automatic Detection of Static and Dynamic Binding

Polymorphism is a means of expressing shared type and is a consequence of supposing systematic sets of relationships among types. Strictly, a polymorphic function is one which may apply to objects of more than one type. Through inheritance, this happens as a matter of course in object-oriented languages. However, the type correctness of expressions involving inherited methods has been questioned [Coo89, CCH89a, CHC90]. The fact that inherited methods change their type has recently been confirmed in [PS94].

In many object-oriented languages, classes are assumed to map directly onto types and a subtyping relationship has to obtain between these types for inherited methods to apply legally [Mey88, SCB86]. This model derives from [CW85] who established the terms *bounded quantification* and *inclusion polymorphism* to describe functions defined over types that are also applicable to subtypes. However, recent research has shown that inheritance is formally not the same thing as subtyping [CHC90]. Some have taken the view that this restricts inheritance to a subsidiary role of sharing implementations [Ame90, Sny87], as

discussed above in section 4. However, we take an alternative view that simple subtyping is too impoverished a type model to cope with what really happens under inheritance.

```
class NUM_COORD[C] : OBJECT[O]
{  data
     x, y : NUMBER[N];
   type
     x () : NUMBER[N];
     y () : NUMBER[N];
     moveTo(NUMBER[N], NUMBER[N]) : NUM_COORD[C];
   ...
}

anIntCoord : NUM_COORD{INTEGER/N};
aRealCoord : NUM_COORD{REAL/N};

class INT_COORD[D] : NUM_COORD[C]{INTEGER/N}

polyIntCoord : INT_COORD[];
genericNumCoord : NUM_COORD;
```

Figure 6: Schema for polymorphism in BRUNEL

In BRUNEL, classes are not types, but rather type constructors, or parameterised polymorphic types. Inheritance is therefore not subtyping, but rather an inclusion relationship among partially-defined type constructors [Sim93]. Polymorphism is indicated in BRUNEL by the presence of type parameters, which brings it much closer to the Milner-style polymorphism of ML [Mil78, MTH90]. Any methods declared over such parameterised types apply legally to all types generated when the parameters are legally instantiated. As a result, polymorphic methods may end up with multiple types and type compatibility is only judged after any remaining type parameters have been replaced.

Figure 6 illustrates the syntactic schema used to describe various kinds of polymorphism in BRUNEL. The NUM_COORD[C] class is like our earlier coordinate class, except that the type of its abscissa and ordinate has not been fixed. The variable *anIntCoord* is of an integer coordinate type generated by replacing the parameter N by INTEGER; and similarly for *aRealCoord*, where N is replaced by REAL. The {x/y} notation indicates parameter substitution and can be used in exactly the same way as the generic mechanisms in Ada and Eiffel. It can also be used to create more restricted classes by partial replacement of parameters, as in the derivation of the class INT_COORD[D]. Note, however, that there is an additional implicit substitution when an actual type is generated from a class. The notation INT_COORD[D] denotes a polymorphic class, whereas INT_COORD denotes the most general type (the *least fixed point*) generated from that class, equivalent to the implicit substitution: INT_COORD[D]{INT_COORD/D}. Type parameters are only strictly required to tie parts of a polymorphic structure together, so that any instantiating type is propagated into all the right parts. So, the notation INT_COORD[] denotes an open-ended class of integer coordinate types, whose type-instantiation is not tied to anything else. The variable *polyIntCoord* has this open-ended set of types and corresponds to the usual notion of a

polymorphic variable in object-oriented languages. The variable *genericNumCoord* on the other hand has a partially-fixed type structure whose internal components have not been resolved. Each of these possibilities present different opportunities (or problems) for optimisation during the binding phase of compilation, when methods are associated with calls.

Conceptually, whenever an object receives a message to perform some action, it responds by finding the nearest appropriate method in the inheritance graph. The task of selecting which method to use in response to a given call is known as *binding* (a function to a symbolic name). If binding happens at compile-time, this is known as *static binding*. If method lookup is delayed until run-time, this is known as *dynamic binding*. If a compiler can determine that only one type of object will ever be present at a given call-site, then it will bind that method statically. Otherwise, if it detects that some small set of object types will be present, then it will bind that method dynamically. Dynamic binding typically arises as a result of a method being *deferred*, ie specified in an abstract class and implemented multiple times in the concrete descendants of that class; or else when a method is redefined multiple times in descendent classes for other reasons, such as for efficiency's sake, or in order to add functionality to the inherited version.

In a strongly-typed language, the great majority of methods to call can be determined statically at compile-time. This is because we know the actual type of the objects to be encountered at the call-site and therefore may rely on the compiler to replace call expressions by function pointers. We have estimated [Sim92] that around 80% of object-oriented code can be bound statically. The speed savings over the dynamic binding approach are considerable: looking up a method takes at least one extra pointer dereference and, on average, 1.3 hashes into a cache, compared with placing a direct call to the method. There are also considerable savings on space, in that any methods which are never invoked dynamically can be removed from run-time dispatch tables (see section 2 for an introduction to dispatching).

For some 20% of object-oriented code, it is essential to have dynamic binding. This is because we do not know the most specific type of the objects encountered at the call-site. Consider a graphics painting package that displays rectangles, circles, triangles etc. and maintains a display list of objects in each screen drawing. This list is a heterogenous collection of graphical objects, whose precise element type is unknown, but which must at least respond to all of the methods of GRAPHIC, a deferred class specifying that all its eventual descendants will implement methods such as *draw*, *rotate* or *fill*. The client code can invoke the *draw* method on each object in the display list, safe in the knowledge that it must provide an implementation of *draw*. However, we cannot detect at compile-time precisely which *draw* method to use in each case. This can only be done by inspecting the type of each object at run-time and looking up the appropriate method.

Untyped languages like Smalltalk [GR83] adopt dynamic binding universally. This is not appropriate for BRUNEL, from either the security or efficiency viewpoints. Objective C [CN91] permits a type-free style of messaging with dynamic binding (such objects are of the type *id*) and, alternatively, a typed style with static binding. C++ and Eiffel have strongly-typed static and dynamic binding, which we adopt as an appropriate starting point for BRUNEL.

Considering the approach taken in C++ [Str91], we find that it is not as flexible or open-ended as it should be. The language forces the programmer to determine in advance whether a method is to be invoked statically or dynamically (dynamic methods have to be declared as *virtual functions*). This is against the spirit of object-oriented programming: if you later wanted to extend the class hierarchy and re-implement a (static) method for dynamic invocation, you would have to return to the original class and change the declaration style of the original method so that it was marked as dynamic. This is wrong firstly in the sense that it requires the breaking and alteration of existing closed modules. Secondly, it provides the wrong solution for the majority of existing applications that do not use the new extension to the class hierarchy and which therefore do not need dynamic binding.

By contrast, Eiffel [Mey92] provides full dynamic binding by default, with a compiler switch to optimise bindings for a given assembly of classes in a post-processing stage. In BRUNEL, we require the compiler to detect the need for dynamic binding automatically as the default option. We can improve upon the speed of Eiffel's post-processing approach because the BRUNEL development environment maintains more information about code as it is developed.

Clearly, it is insufficient to rely on some labelling scheme (such as *virtual* or *deferred*) to trigger the appropriate choice of dynamic over static binding, since merely re-implementing a method in a descendant may entail a need for dynamic binding. In its binding phase, the BRUNEL optimiser performs the following steps:

a) A complete subgraph representing the portion of the developer's class graph used in the application is constructed and collapsed using the optimisations outlined in steps 4 (a) - 4 (g) above. This eliminates spurious sibling and cousin classes which are not used in the application, flattens chains of intermediate ancestors down to shared branch-points, retaining these and all classes which are eventually instantiated in the application. Each class in the collapsed graph obtains a static type signature for every method that it understands, including inherited methods with re-bound type parameters.

b) The method which initiates the application becomes the root of a call-graph which is constructed by tracing all nested invocations of methods in the application program. Each call is a node in this graph, labelled with a most specific static type, obtained by consulting the type schema of the receiver object (the target variable of the call). Two invocations of the same method may therefore have different static type, depending on the type of the receiver at the call-site. Any two such distinct invocations become separate nodes in the graph. On the other hand, nodes generated at different places, but which are labelled with identical method names and types, become merged. Recursive methods therefore generate self-transitions and mutually recursive methods generate closed loops.

c) For each type-distinct method invocation, the inheritance graph is searched from the class of the target object downwards towards its leaves (ie in the reverse direction to normal inheritance), in order to determine if there is a re-implementation of the method in any of the application classes. The number of re-definitions is logged and the methods are marked for inclusion.

d) For each type-distinct method invocation, the inheritance graph is then searched upwards from the class of the target object until a declaration or implementation for the

method is found. The most specific declaration or implementation is logged and marked for inclusion.

e) Each type-distinct method invocation is now marked for static or dynamic binding. If the call has a single implementation, this is bound statically. If the call has more than one associated implementation, all these are included in a dispatch table and the method is bound dynamically. Dynamic binding will result either from a method being deferred with multiple definitions, or from a method being implemented but having at least one re-definition. A deferred class with a single implemented descendant will of course have been "flattened" in step (a) above.

f) Any methods which are not marked in the class hierarchy during the traversal of the call graph are never used in the application; they can be eliminated from the target program. This task should really be delegated to a smart linker (see section 2); in the absence of such, and since we are performing a cross-translation to C, we do it here.

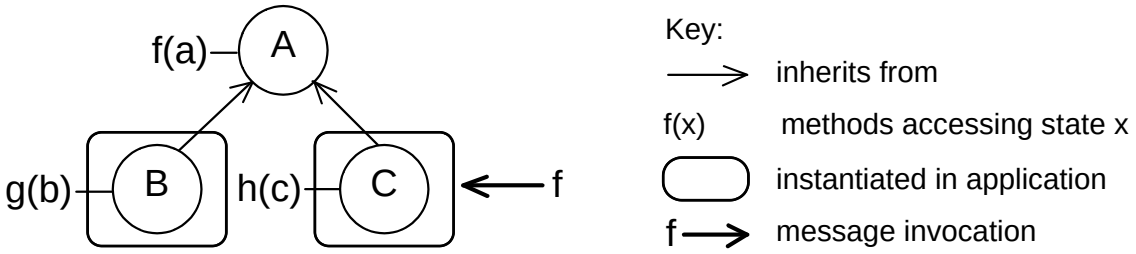

Figure 7a: Static binding for a single method

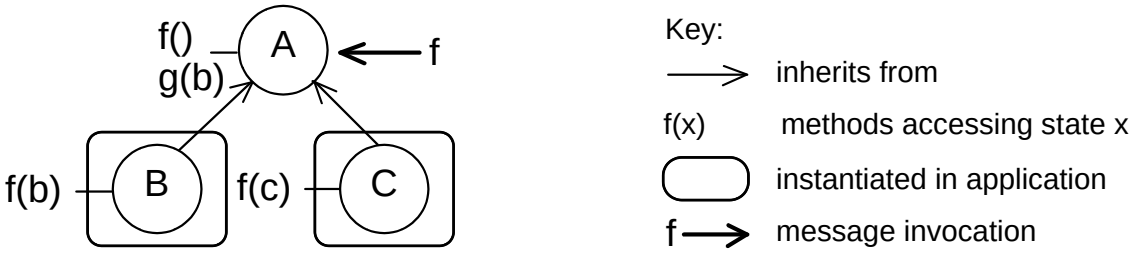

Figure 7b: Dynamic binding for a deferred method

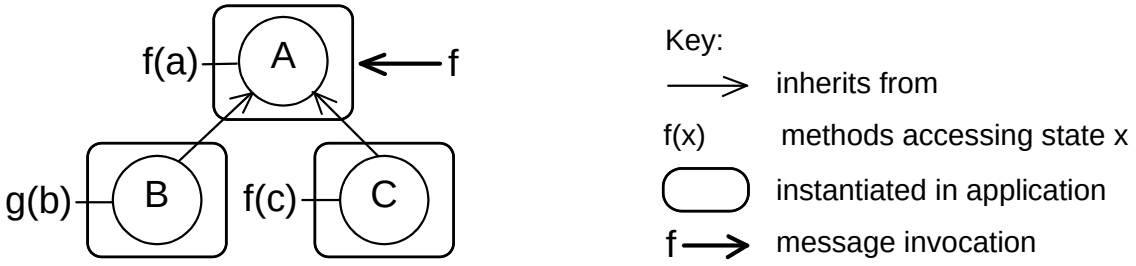

Figure 7c: Dynamic binding for a redefined method

Figures 7a - 7c, adapted from [Ng92], illustrate the distinct cases in determining static and dynamic binding. Figure 7a illustrates the case where only one static method can be found

for a class. Figure 7b illustrates the case where a deferred method has two implementations in different branches and therefore is bound dynamically. Figure 7c illustrates the case where an implemented method has been redefined in a descendent class, therefore dynamic binding is required.

Our scheme as described can be applied as part of a global binding optimisation strategy in any strongly-typed object-oriented language. Because we express it in terms of traversing a call-graph, we ensure that different invocations of the same method are not lumped together in the analysis of binding. A more approximate local analysis would only consider each method once for possible re-definitions in the class hierarchy, whereas our approach takes into account each call and the most specific type of the target variable at the call-site. In consequence, this scheme allows the same method to be statically bound over some variables but dynamically bound over others, where it is subject to re-implementation. It all depends on the branch of the inheritance graph in which the target of the invocation is typed. Note finally how this scheme allows binding to depend on which classes are needed for an application, such that incremental extensions to the developer's class library which would require dynamic binding do not enforce this upon existing applications.

It is clear that this optimisation is a global post-process for specific applications. Like the inheritance optimisations described in section 4, it could not easily be applied in an incrementally compiled regime. For the moment, the language definition of C++ rules out the automatic determination of binding anyway. Eiffel optionally optimises global system bindings as a post-process, but this takes up a considerable amount of extra time. Instead, we suggest that a scheme similar to the one we described in section 4 for the early detection of stable inheritance subsystems in a given application might be explored. If the dependency information recorded for a class under its <name>.E directory were to include counts of method redefinitions and were to log the types that were the targets of creation instructions below it in the inheritance hierarchy, then stable binding subsystems could be established during the development of an application. This would allow the early generation of object-code in which bindings were optimised. Local changes to classes would have to be propagated upwards through the dependency files of ancestor classes. A sensitive change would result in the recompilation of any client invoking a method whose binding status had altered, the next time the system was built.

Curiously, hardly any reported research seems to have adopted our emphasis of detecting dynamic binding automatically in a statically-typed regime. We assume that strongly-typed languages like Eiffel, Trellis, POOL-I and CommonObjects must use a similar approach, but the techniques used have not been widely reported. By contrast, a lot of recent work has focussed on opportunities for detecting early static binding in a dynamically-typed regime.

TS (standing for Typed Smalltalk) [JGZ88, GJ90] is geared towards retrofitting a full static type system for Smalltalk, with optimisation in mind. The type system for TS is different from BRUNEL, being independent of the class hierarchy and based on discriminated unions and signature types. Like the separate *protocols* of Objective C [NeX93], this is necessary to capture the common type of features introduced at disjoint parts of the (single) inheritance hierarchy. Type annotations are provided and some type inference is performed, permitting the early binding of many methods.

Perhaps the most interesting sequence of work on binding optimisation however has been carried out by the designers of Self [CUL89, CU90, CU91, CUL91]. Self has no classes (it is

prototype-based) and no type annotations; furthermore it has user definable control structures and dynamic inheritance, making the optimisation task even more challenging. To compensate for the lack of class types, the Self compiler builds implementation-level maps to group objects cloned from the same prototype. Then, methods are dynamically compiled as they are first invoked. Through the techniques of *type prediction* and *message splitting*, multiple versions of a method are compiled, some versions optimised for particular common types (eg integer, boolean objects). Within each version, the type of the receiver is fixed and this enables the static analysis of further nested calls. The effect of this is to reduce dynamic dispatching considerably; many nested calls are simply inlined.

Our approach shares the goals of these examples, in that nowhere in the language syntax should the full potential for dynamic use be compromised. Since BRUNEL aims to reduce the size of object-code, we refrain from the extended message-splitting techniques of Self, as this increases the number of recompilations of methods, which is exacerbated in a finely-factored multiple inheritance regime. The trade-off in Self is between losing dispatching code and gaining multiple copies of other operational code. In BRUNEL, we currently choose to ignore some of the opportunities for early static binding because of this code size trade-off. Consider Smalltalk's abstract class Collection, whose methods *collect*: and *select*: capture the significant abstraction of mapping and filtering over collections; or *addAll*: which appends all the elements of one collection to another. These methods are implemented in terms of the more primitive iteration method, *do*:, and the method to add single elements, *add*:, which are deferred and implemented multiple times, once for each kind of collection. Calls to *do*: and *add*: are dynamically bound and dispatched inside these methods. Now, if there were five significantly different subclasses of Collection and we wished to optimise the bindings of *do*: and *add*: such that they were bound statically, this would yield a five-fold increase in Collection's methods, in terms of recompiled versions of *collect*:, *select*: and *addAll*:. In BRUNEL, we clearly do not want an arbitrary geometric increase in object-code size. Instead, we monitor the number of retyped versions of methods at polymorphic call-sites and can set a threshold to limit the number of recompilations, reverting to dynamic binding if this is exceeded.

Our scheme produces the tightest analysis of binding for a language like BRUNEL. This is because BRUNEL's type system removes ambiguities present in other strongly-typed object-oriented languages. In Eiffel, a compiler encountering a variable with the type annotation GRAPHIC cannot tell whether this refers to the precise *type* of object to be stored there, or a polymorphic *class* of object-types to be stored there. This is immediately apparent in BRUNEL, which makes the distinction between GRAPHIC, an albeit rather general *type*, and GRAPHIC[], a polymorphic *class*. Annotations that resolve to types can be bound statically without further consideration. Polymorphic annotations like GRAPHIC[G] may or may not result in dynamic binding. This is because type constraints can be propagated through the call graph, especially where function results become targets for further calls. An expression containing a polymorphic call to *draw* might already have replaced a parameter tied to G by an actual type, say CIRCLE, for that invocation of the function. As a result, that call could be bound statically, although in general *draw* was designed for polymorphic invocation.

BRUNEL's type parameters can be thought of as a properly worked out alternative to Eiffel's *like <anchor>* mechanism, in which the type of an expression can evolve under inheritance or as a result of combinations with other types [PS94]. In Eiffel, *like Current* expresses the type-recursion of the self-type under inheritance. BRUNEL captures this with the parameter

O in OBJECT[O]. Whereas Eiffel has a mixture of anchored types, ambiguous mono/polymorphic types and constrained generic types, all of which interact, BRUNEL has one single scheme, in which type parameters are replaced either at compile time or run time. Furthermore, the unification mechanism for resolving the types of polymorphic expressions is already well-understood from ML [Mil78].

6. Removing Levels of Nesting in Call Graphs

In BRUNEL inheritance is seen firstly as a sharing of behavioural specification and only secondly as a sharing of implementation. BRUNEL has no need to divorce the type hierarchy completely from the class hierarchy [Ame90, Sny87] since inheritance is not subtyping; instead it involves the substitution of type parameters to create functions of different, but related, types. Type consistency is only considered when all parameters are replaced; and so a polymorphic method may turn out to have a set of types, corresponding to a set of valid instantiations of type parameters, and one of these types will apply correctly to each descendant of the class in which the method was declared.

However, insisting on type-consistent inheritance restricts the number of ways in which class implementations can physically be altered. Inheritance graphs in BRUNEL typically consist of many small clusters of closely related types, rather than extended trees. As a result, we expect that code reuse in BRUNEL will depend more on composition than on the kinds of opportunistic inheritance of implementations practised in languages like Smalltalk. For comparison, an even more extreme reaction against implementation inheritance is found in Emerald [RL89] where all implementation reuse is achieved through composition. This has the potential problem of introducing more levels of embedding in structures and more levels of nesting in call-graphs than in some object-oriented languages.

To explain how this arises, consider that classes do not allow unrestricted access to their internals - this is managed by their interface, expressed as a set of access and update methods. The clear advantage of the protection offered by encapsulation is then somewhat tempered when you consider that requests directed at a deeply embedded class have to penetrate several abstraction barriers, and such a class can only communicate back via a chain of nested method calls. By contrast, the attraction of using inheritance in the opportunistic sense lies in the direct applicability of inherited methods to the inheriting class, which also has the right to access and set attributes of its inherited part directly. However, BRUNEL rejects opportunistic inheritance where this violates strict type discipline.

As an example, consider a language with non-strict inheritance which permits RECTANGLE to be a child class of POINT. The idea is that POINT has two attributes, *x* and *y*, which store its screen location. RECTANGLE adds two more attributes, *length* and *width*, while inheriting the two coordinates of its origin. This is manifestly good from the point of view of economy of implementation - to move the rectangle around on the screen, you need only invoke the *move* method inherited from POINT which will also update the appropriate attributes in RECTANGLE objects. On the other hand, it is patently ridiculous from the point of view of types. A rectangle is not really a kind of point, in terms of geometrical classification, but rather some kind of quadrilateral, along with rhombus and square. It is more reasonable to suggest that a rectangle is *composed of* points (it can be constructed in several ways). If we construct RECTANGLE as a structure containing a POINT object and two INTEGER objects, we would necessarily require RECTANGLE to have its own *move*

method, which in turn would invoke the *move* method of the POINT class. While RECTANGLE's method rightly expresses something about the specification of rectangles, it merely serves to invoke another method which does the job. When this problem is scaled up to embedded, multi-component classes, in the worst case there may be several layers of method calls, simply conveying requests from the outside world to the most embedded components.

Since BRUNEL's priority is to maintain type discipline and encapsulation at all costs, it is right that at the programmer-interface level RECTANGLE is seen to have a *move* method of its own, since this is a necessary part of its specification (in fact, possibly shared by all GRAPHIC objects). This does not mean to say that a compiler could not replace calls to this interface method by something more efficient. Here, BRUNEL adopts an aggressive inlining strategy, automatically removing methods which are present in the specification of classes in favour of something more efficient in cases where this is absolutely safe. For example, the following calls, once it were established that they were legal and safe, could be replaced:

a) access methods whose sole purpose is to deliver the value of a constant attribute (shared attribute) may be replaced by the value of that attribute;

b) access methods whose sole purpose is to deliver the value of an attribute may be compiled out to an offset into structure;

c) assignment methods whose sole purpose is to store a value in an attribute may be compiled out to a remote assignment to an offset into a structure;

d) methods which are short and non-recursive may be replaced inline by their body, provided that they can be statically bound;

e) methods which invoke primitive machine instructions, such as integer arithmetic, comparisons and array access, may be replaced inline by their hard-wired definitions;

f) calls to interfacing methods whose sole purpose is to invoke another method with the same arguments may be replaced inline by a remote call to the enclosed method applied to the appropriate offset of the enclosing structure.

Our call-replacing strategies are inspired by Self's *automatic message inlining* [CUL89] in steps (a) - (e). As in Self, we lay emphasis on the automatic detection of safe cases for inlining and do not currently aim to include a special *inline* directive in the language, as found in C++. Special treatments of reader/writer methods are already given in other languages, for example the fact that all reader/writer methods in CLOS [Kee89] convert at the system level to the *slot-value* function allows compilers to replace this if desired. The last form of inlining (f) is a special case of (d) brought on by the need to maintain strict encapsulation in BRUNEL and leads to a dramatic reduction in levels of nesting in call-graphs for deeply buried objects. The argument for inline replacement is especially clear, since the nested method has the same number and types of arguments as the enclosing method. It may even have the same name, as in the case of *move* above. This stage alone achieves automatically many of the common inlining techniques used by C++ programmers, for example, when overloading the array access operator [ ] for properly-encapsulated vectors.

The current implementation of BRUNEL only supports the inlining steps (a) - (c) and (f), since we have not yet determined the safety of replacements in the other cases. It is clear that any scheme should prohibit the wholesale replacement of a class's interface by inline calls. Wherever dynamic binding is detected, the dispatching call would still have to remain; however inlining might still be done in individual variants of a method. There are two further impediments to the successful application of inlining step (d). If the body of such a method repeats any of the argument expressions, the compiler would have to determine whether this was in fact a single object reference or a sub-expression. If the latter, further steps would have to be taken to eliminate the repeated evaluation of sub-expressions. This might become part of a more general peep-hole optimisation strategy. Furthermore, the wholesale inlining of methods has a tendency to increase the size of the object-code. Our step (f) even prohibits the transformation of arguments before passing them on to the nested method, such that we can guarantee a reduction in the size of the object-code. For the more general inlining of (d) we should have to perform further tests to establish an optimum size for methods to be inlined. We have not yet addressed step (e) since the prototype implementation of BRUNEL currently cross-compiles to C-code which we are using as a portable pseudo-assembler. A future native code compiler would seek to exploit this inlining step.

7. Conclusions

We have presented several strategies for analysing and optimising object-oriented software in BRUNEL. Such stragegies may be useful in any object-oriented language with strong static typing. The optimisations are specific to given delivery systems and are aimed at reducing the size of the object-code module, while achieving all speed performance advantages which do not mitigate against this.

In our modelling experiments [Low91], we have demonstrated in practice that it is possible to break cycles of dependency among classes, that inheritance graphs can be collapsed without loss of semantics and that binding can be determined automatically. Two compilers for different prototype versions of the language have been written so far [Ng92, Bla92] which transform BRUNEL sources into a C-based pseudo-assembler.

The best-case performance of our compilation model involves a memory-transfer from the development environment of parts of the class dependency graph used in the application and a single-pass recursive descent parse of class implementations. The worst-case performance involves a two-phase approach. In the first phase, the dependency graph is constructed by "peeking" at the interface portion of source files and the second phases is as above.

The collapsing of the inheritance graph is an innovation in object-oriented optimisers and contributes to the reduction in the size of the object-code. Programmers may freely exploit finely-factored multiple inheritance without penalty. The automatic detection of static and dynamic binding removes the need for *virtual* directives and allows methods which are bound statically in one application to be bound dynamically in another, according to need. Furthermore, the same method may be bound statically or dynamically at different call-sites in a single application. A series of call-inlining strategies were also discussed. The emphasis in BRUNEL is to provide automatic inlining where this is guaranteed to reduce the size of the object-code. Specific attention was drawn to a technique for removing levels of indirection in call-graphs forced by the requirements of strict encapsulation.

References

[Ame90] P America (1990), 'Designing an object-oriented programming language with behavioural subtyping, *Proc. Conf. on Foundations of Object-Oriented Languages*, 60-90.

[Bla92] D Black (1992), 'An optimising compiler for an object-oriented language with higher-order functions', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[BS83] D Bobrow and M Stefik (1983), *The LOOPS Manual*, Xerox Corporation.

[CCH89a] P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for object-oriented programming', *Proc. 4th Int. Conf. on Functional Prog. Lang. and Comp. Arch.*, Imperial College, 11-13 September, 273-280.

[CCH89b] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly typed object-oriented programming', *Proc. ACM Conf. on OOPSLA*, 457-467.

[CHC90] W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. ACM Symp. on Principles of Programming Languages*, 125-135.

[CN91] B J Cox and A J Novobilski (1991), *Object-Oriented Programming: an Evolutionary Approach*, 2nd. Edn., Addison Wesley.

[Coo89] W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. European Conf. on Object-Oriented Programming*, 57-70. Also pub. *British Computer Journal 32 (4)*, 305-311.

[CU90] C Chambers and D Ungar (1990), 'Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs', Proc. ACM SIGPLAN Conf. on Programmng Language Design and Implementation, pub. *SIGPLAN Notices 25 (6)*, 150-164.

[CU91] C Chambers and D Ungar (1991), 'Iterative type analysis and extended message splitting: optimizing dynamically-typed object-oriented programs', *Lisp and Symbolic Computation 4 (3)*, 283-310.

[CUL89] C Chambers, D Ungar and E Lee (1989), 'An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes', Proc. ACM Conf. on OOPSLA, pub. *SIGPLAN Notices 24 (10)*, 49-70

[CUL91] C Chambers, D Ungar and E Lee (1991), 'An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes', *Lisp and Symbolic Computation 4 (3)*, 243-281.

[CW85] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys 17 (4)*, 471-522.

[GJ90] J O Graver and R E Johnson (1990), 'A type system for Smalltalk', *Proc. ACM Symp. on Principles of Programming Languages*, 136-150.

[Gol85] A Goldberg (1985), *Smalltalk-80: the Interactive Programming Environment*, Addison Wesley.

[GR83] A Goldberg and D Robson (1983), *Smalltalk-80: the Language and its Implementation*, Addison Wesley.

[How93] R Howard (1993), 'Eiffel: an overview', *Journal of Object-Oriented Programming, 5 (8)*, January, 76-79.

[JGZ88] R Johnson, J Graver and L Zurawski (1988), 'TS: an optimizing compiler for Smalltalk', *Proc. ACM Conf. on OOPSLA*, 18-25.

[Kee89] S E Keene (1989) *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*, Addison Wesley.

[Low91] Low E K (1991), 'A class structure optimiser for object-oriented languages', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[Mad93] O L Madsen (1993), *Object-Oriented Programming in the Beta Programming Language*, Addison Wesley.

[MBD90] B Magnusson, M Bengtssson, L-O Dahlin, G Fries, A Gustavsson, G Hedin, S Minor, D Oscarsson and M Taube (1990), 'An overview of the Mjølner/ORM environment: incremental language and software development, *Technical Report LU-CS-TR:90-57* and *LUTEDX/(TFCS-3026)/1-12/(1990)*, Department of Computer Science, Lund University, Lund Institute of Technology, Sweden.

[Mey88] B Meyer (1988), *Object-Oriented Software Construction*, Prentice-Hall.

[Mey92] B Meyer (1992), *Eiffel: the Language*, Prentice Hall.

[Mil78] R Milner (1978), 'A theory of type polymorphism in programming', *J. of Computer and Systems Sciences 17*, 348-375.

[MTH90] R Milner, M Tofte and R Harper (1990), *The Definition of Standard ML*, MIT Press.

[NeX93] NeXT Computer Inc (1993), *NeXTStep Object-Oriented Programming and the Objective C Language*, Addison Wesley.

[Ng92] Ng Y M (1992), 'An optimising compiler for handling multiple inheritance in BRUNEL', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[Ong91] Ong P S (1991), 'An object-oriented graphical user interface for BRUNEL', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[PS94] J Palsberg and M Schwartzbach (1994), *Object-Oriented Type Systems*, John Wiley.

[Qui90] B Quinn (1990), 'Object-oriented languages for software engineering: a comparative study', *unpublished MSc dissertation*, Department of Computer Science, University of Sheffield, UK.

[RL89] R K Raj and H M Levy (1989), 'A compositional model for sofware reuse', *Proc. European Conf. on Object-Oriented Programming*, 3-24; also pub. *British Computer J. 32 (4)*, 312-322.

[SC92] A J H Simons and A J Cowling (1992), 'A proposal for harmonising types, inheritance and polymorphism for object-oriented programming', *Dept. Comp. Sci. Research Report CS-92-13*, University of Sheffield, UK.

[SCB86] C Schaffert, T Cooper, B Bullis M Kilian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. ACM Conf. on OOPSLA*, 9-16.

[Sim91] A J H Simons (1991), 'BRUNEL: a strongly-typed, portable object-oriented language and programming environment', *Dept. Comp. Sci. Research Report CS-91-07*, University of Sheffield, UK.

[Sim92] A J H Simons (1992), 'Using and teaching object-oriented languages: the future of software engineering', *Proc. 2nd Nat. Conf. on Soft. Eng. in Higher Education*, Southampton Institute, 23-34.

[Sim93] A J H Simons (1993), 'Introduction to object-oriented type theory', *ACM Conf. on OOPSLA-93 Tutorial Notes*, 62pp.

[Sny87] A Snyder (1987), 'Inheritance and the development of encapsulated software components', *Proc. 20th Hawaiian Int. Conf. Sys. Sciences, Vol 2*, 227-238; also in B Shriver and P Wegner (eds), *Research Directions in Object-Oriented Programming*, MIT Press, 165-188.

[Str91] B Stroustrup (1991), *The C++ Programming Language*, 2nd. Edn., Addison Wesley.

[Tam92] R Tam (1992), 'A multiple inheritance graphical browser', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[Tse91] Tse H P (1991), 'A class structure- and a formal type-checker for an object-oriented language', *unpublished MSc dissertation*, Dept. Comp. Sci., University of Sheffield, UK.

[Wir82] N Wirth (1982), *Programming in Modula-2*, Springer Verlag.