Inheritance Management and Method Dispatch in Reflexive Object-Oriented Languages

Wade Holst Duane Szafron

December 27,1996

Abstract

A collection of algorithms and data structures are presented which represent a generalized framework for inheritance management and method dispatch in reflexive, dynamically typed, single-receiver languages with type/implementation-paired multiple inheritance. By storing a small amount of information, the algorithms can incrementally maintain the entire dispatch environment during the four fundamental environment modification requests – adding/removing selectors to/from classes and adding/removing class hierarchy links. By merging inheritance management, inheritance conflict detection and method dispatch calculation, the algorithms are computationally efficient, and can be used to maintain dispatch information even in a reflexive environment. The algorithms are applicable to all table-based method dispatch techniques and require the use of only a few dispatch-specific functions. A group of object-oriented classes are used to implement both the technique-independent and technique-dependent algorithms, providing a complete framework for table-based method dispatch. Although general enough to apply to reflexive languages, the framework is also useful in statically typed languages, as it incrementally computes hierarchy information needed by the compiler to establish which method addresses can be uniquely identified during compilation. This allows compile-time optimizations instead of a runtime table look-up. The framework can and will be extended to multi-method languages.

1 Introduction

Object-oriented programming languages have become popular due to the abstraction and information hiding provided by inheritance and polymorphism. However, these same properties pose difficulties for efficient implementation, necessitating (among others) algorithms for inheritance management and method dispatch. In this paper, we present an object-oriented solution to an object-oriented problem.

Various object-oriented programming techniques, called *design patterns*, have been identified. One such pattern is the *template method*, which "Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses" ([GHJV95]). The first part of this paper presents a collection of such template methods that identify all actions necessary for any table-based method dispatch technique to incrementally maintain a dispatch table. Each of these methods is technique-independent, calling technique-dependent operations to provide low-level functionality like table access and modification.

Object-oriented languages provide code-reuse at two levels. At the first level are generic libraries of basic data structures like sets and growable arrays. Rich libraries for collections, graphics and other specialized areas provide object-oriented languages with much of their power. At a second level, *application frameworks* capture the collaborations of a group of objects, leaving the specific details to be implemented. These details are implemented by framework clients, who subclass on the classes provided by the Framework. These subclasses implement concrete versions of the abstract functionality to provide client-specific behavior. In other cases, the user merely chooses between concrete leaf classes to obtain the desired functionality. Thus, in the same way that C++ *templates* generalize the implementation of a particular class, *frameworks* generalize the implementation of an entire group of interacting classes. Templates are instantiated by providing parameters to the template class. Frameworks are instantiated by providing concrete implementations of abstract functions. The second part of this paper presents the DT Framework; a general framework for both compile-time and run-time inheritance management and method dispatch.

The DT algorithms (and hence the DT Framework) apply to a broad class of object-oriented languages: reflexive, dynamically typed, single-receiver languages with type/implementation-paired multiple inheritance. A *reflexive* language is one with the ability to define new methods and classes at run-time. A *dynamically typed* language is one in which some (or all) variables and method return values are unconstrained, in that they can be bound to instances of any class in the entire environment. A *single-receiver* language is one in which a single class, together with a selector, uniquely establishes a method to invoke (as opposed to multi-method languages, discussed in Section 8). *Type/-implementation-paired inheritance* refers to the traditional form of inheritance used in most object-oriented languages, in which both the definition and implementation of inherited selectors are propogated together (as opposed to inheritance in which these two concepts are separated, as discussed in Section 8). Finally, *multiple inheritance* refers to the ability of a class to inherit selectors from more than one direct superclass. Within this paper, we will refer to this collection of languages as Ψ.

Any compiler or run-time system for a language in Ψ must implement the functionality provided by the DT algorithms. Furthermore, since the DT algorithms merge inheritance propogation and dispatch-table modification, they are highly efficient. As well, the DT Framework, built on top of the DT algorithms, provides implementors of languages in Ψ with immediate code reuse. In this paper, we will refer to compilers and run-time systems as DT Framework clients. For our purposes, a language that can be compiled is inherently non-reflexive, and *compilers* can be used on such languages. By *run-time system* we mean support existing at run-time to allowing reflexivity in the language.

This paper makes a variety of research contributions. It extends research in each of these areas:

1. *Framework*: the algorithms can be implemented as a collection of classes. Framework clients (implementors of languages in Ψ) obtain efficient inheritance management, inheritance conflict detection, and incremental dispatch-table modification (as well as other advantages) via inheritance.

2. *Data Structures*: the *division* data structure is identified, a critical structure that allows inheritance management to be made incremental, allows detection and recording of inheritance conflicts, and maintains information useful in compile-time optimizations.

3. *Algorithms*: The framework demonstrates how inheritance management and maintenance of dispatch information can be made incremental. A critical recursive algorithm is designed that handles both of these issues and recomputes only the information necessary for a particular environment modification. As well, the similarities and differences between adding information to the environment and removing information from the environment are identified, and the algorithms are optimized for each.

4. *Table-Based Dispatch*: The framework identifies the similarities and differences between the various table-based dispatch techniques. It shows how the division data-structure and inheritance management algorithms can be used to allow incremental modification of the underlying table in any table-based dispatch technique.

The division data structure, the incremental algorithms, and their ability to be used in conjunction with any table-based dispatch technique results in a complete framework for inheritance management and maintenance of dispatch information that is usable by both compilers and run-time systems. The algorithms provided by the framework are incremental at the level of individual *environment modifications*, consisting of any of the following:

1. Adding a selector to a class.

2. Adding one or more superclasses to an existing class (which may already have zero or more superclasses).

3. Removing a selector from a class
4. Removing one or more superclasses of an existing class.

The following capabilities are provided by the algorithms:

1. *Inheritance Conflict Detection*: In multiple inheritance, it is possible for inheritance conflicts to occur when a selector is visible in a class from two or more superclasses. The Framework detects and records such conflicts as they occur.

2. *Dispatch Technique Independence*: Clients of the framework can change between different table-based dispatch techniques with a simple recompilation.

3. *Reflexive Languages*: Dispatch tables have traditionally been created by compilers and were not extendible at run-time. This implied that reflexive languages could not use table-based dispatch techniques. By making dispatch table modification incremental, the DT Framework allows reflexive languages to use any table-based dispatch technique, maintaining the dispatch table at run-time as the environment is dynamically altered.

4. *Dynamic Schema Evolution*: The DT Framework provides efficient algorithms for arbitrary environment modification, including adding class hierarchy links to classes already in an inheritance hierarchy. Even more important, the algorithms handle both additions to the environment *and* deletions from the environment.

5. *Separate Compilation*: Of the five table-based dispatch techniques discussed in Section 2, four of them currently require knowledge of the complete environment. In situations where library developers provide object files, but not source code, these techniques are unusable. Incremental dispatch table modification allows the DT Framework to provide separate compilation in all five dispatch techniques.

6. *Compile-time Method Determination* : It is often possible (especially in statically typed languages) for a compiler to uniquely determine a method address for a specific message send. The more refined the static typing of a particular variable, the more limited is the set of applicable behaviors when a message is sent to that variable. If only one method applies, the compiler can generate a function call or inline the method. The division data structure maintains information to allow immediate determination of such uniqueness.

The rest of this paper is organized as follows. Section 2 summarizes the various method dispatch techniques. Section 3 presents the DT algorithms. Section 4 presents the DT Framework. Section 5 discusses how the table-based method dispatch techniques can be implemented using the DT Framework. Section 6 discusses details specific to compilers and details specific to run-time systems. Section 7 reports execution performance results when the DT Framework is applied to various real-world class hierarchies. Section 8 discusses related and future work. Section 9 summarizes the results. Appendix A provides proofs of some assertions, and Appendix B provides utility algorithms. Both the assertions and utility algorithms are used in Section 3. Finally, Appendix C describes how the information stored by the DT algorithms can be used to establish whether a method is uniquely determined at compile-time.

2 Method Dispatch Techniques

Due to inheritance and polymorphism, it is not possible to always determine the method to be invoked at a particular call-site at compile-time, so some run-time method dispatch technique is necessary. Two primary categories of method dispatch techniques exist: dynamic techniques and table-based techniques. This paper generalizes existing research into table-based dispatch techniques. Each table-based technique is described in detail in the subsections that follow. However, a very brief description of the various dynamic dispatch techniques is provided first, where C denotes a class and σ a selector.

ML) *Method Lookup*[1] (Smalltalk-80 [GR83]). Method dictionaries are searched for selector σ starting at class C, going up the inheritance chain, until a method for σ is found or no more parents exist (in which case a *messageNotUnderstood* method is invoked to warn the user). This technique is space efficient but time inefficient.

LC) *Global Lookup Cache* ([GR83, Kra83]) uses $< C, \sigma >$ as a hash into a global cache, whose entries store a class C, selector σ, and address A. During a dispatch, if the entry hashed to by $< C, \sigma >$ contains a method for the class/selector pair, it can be executed immediately, avoiding ML. Otherwise, ML is called to obtain an address and the resulting class, selector and address are stored in the global cache.

IC) *Inline Cache* ([DS94]) caches addresses at each call-site. The initial address at each call-site invokes ML, which modifies the call-site once an address is obtained. Subsequent executions of the call-site invoke the previously computed method. Within each method, a *method prologue* exists to ensure that the receiver class matches the expected class (if not, ML is called to recompute and modify the call-site address).

PIC) *Polymorphic Inline Caches* ([HCU91]) cache multiple addresses, modifying a special call-site specific stub-routine. On the first invocation of a stub-routine, ML is called. However, each time ML is called, the stub is

[1]In [DHV95, Dri93a], and others, this is referred to as Dispatch Table Search (DTS). However, to avoid confusion with our dispatch tables, we refer to it as Method Lookup

extended by adding code to compare subsequent receiver classes against the current class, providing a direct function call (or even code inlining) if the test succeeds.

In subsections that follow, each of the table-based techniques are presented and discussed in detail. We will use definitions and notations from Table 1 during the discussion, and will provide example dispatch tables based on the inheritance graph in Figure 1. The exact structure of the dispatch table depends on the dispatch technique. In our discussion, we will represent the tables as global two dimensional tables. However, in an implementation, it is not strictly necessary, and sometimes not desirable, to have global tables, since per-selector or per-class arrays can improve data locality. In all of these techniques, classes and selectors are assigned numbers which serve as indexes into the dispatch table. We have arbitrarily choosen to index rows by selectors and columns by classes, and to treat tables as row-major. In the tables displayed, the notation C:σ is used to refer to the method that is defined natively in class C for selector σ. If C:σ exists as an entry for some subclass, C_i of C, it implies that C_i inherits σ from C.

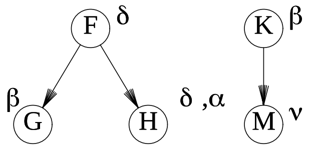


Figure 1: Sample Inheritance Graph

In developing the DT algorithms, we have abstracted out those concepts that are similar among the various table-based dispatch techniques. Each technique has an underlying *data* array, and uses selector and class indices to obtain the address for a particular class/selector pair. Each selector and class is assigned an *index*. Each class/selector pair, $< C, \sigma >$, establishes an index pair $< L, K >$ which is used to determine a unique index, T$[\sigma, C]$, within the underlying *data* array of the table in a technique-dependent fashion. Each entry in the table contains a *division*. A division represents a method to be executed for a particular class/selector pair, but contains extra information in addition to the method address (usually, a class and a selector). The class associated with a division is referred to as the *defining class* of the division. *Selector index conflicts* can occur in certain dispatch techniques, when T$[\sigma, C]$ returns a division that does not represent selector σ. *Class index conflicts* are also possible, occuring when T$[\sigma, C]$ does not represent class C.

While describing the table-based method dispatch techniques, we will present simple high-level algorithms describing how the dispatch techniques can be implemented if complete knowledge of the environment exists. We will then discuss

Notation	Definition
σ	a selector
L	a selector index
index(σ)	the current index of selector σ
K	a class index
C,C_i	classes
$C_i < C$	class C_i is a subclass of class C
$< C, \sigma >$	notation to represent a class/selector pair
index(C)	the current index of class C
subclasses(C)	the set of all subclasses of C
children(C)	the set of immediate subclasses of class C
parents(C)	the set of immediate superclasses of class C
selectors(C)	the set of selectors defined natively in C
T	a table
T[L, C]	the table entry identified by selector index L and class C
T[σ, C]	short-hand for T[$index(\sigma), C$]
methodFor(σ,C)	the method to execute for class C and selector σ

Table 1: Notations and Definitions for the DT algorithms

how these algorithms need to be modified to allow them to be incremental. For our purposes, an *incremental algorithm* for dispatch-table maintenance is one that modifies an existing dispatch table each time an environment modification occurs (adding/removing a selector or hierarchy link). The algorithm must work independent of the order in which environment modifications occur (although differing orders may produce better compression results and execution times).

2.1 Selector Table Indexing (STI)

Selector Table Indexing ([Cox87]) is the most time efficient, but space-inefficent, table-based dispatch technique. It uses a two-dimensional table in which both class and selector indices are unique. Even in dynamic languages where it is possible to invoke a non-understood message, no special code is necessary; the dispatch table stores the address of a special error method for any class/selector pairs that do not have an associated method. Unfortunately, although this approach is fast, it is not feasible for even medium sized environments because the space required is the product of the number of classes and selectors. Table 2 shows an STI dispatch table for Figure 1.

selectors	**index**	**F**	**G**	**H**	**K**	**M**
δ	0	F:δ	F:δ	H:δ	-	-
β	1	-	G:β	-	K:β	K:β
α	2	-	-	H:α	-	-
ν	3	-	-	-	-	M:ν

Table 2: STI dispatch table

A simple, efficient algorithm to assign class and selector indices is easily implemented. Since class and selector indices are unique and orthogonal to one another, the algorithm works equally well in either an incremental or non-incremental setting.

```
Algorithm STI
    L := -1; K := -1
    foreach class C
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            if index($\sigma$) is unassigned
                L := L+1
                index(S) := L
            T[L][K] := methodFor($\sigma$,C)
end STI
```

2.2 Selector Coloring (SC)

Selector Coloring ([DMSV89, AR92]) compresses the two-dimensional STI table by allowing selector indices to be non-unique. Two selectors can share the same index as long as no class recognizes both selectors. The amount of compression is limited by the largest complete behavior (the largest set of selectors recognized by a single class). Since this approach is implementable as a graph coloring algorithm, the selector indices are usually referred to as *colors*.

Table 2 can be colored to produce Table 3. Since no class understands both α and β, the rows for these two selectors can be merged into one. Similarily, the rows for δ and ν can also be merged.

selectors	**index**	**F**	**G**	**H**	**K**	**M**
δ, ν	0	F:δ	F:δ	H:δ	-	M:ν
α, β	1	-	G:β	H:α	K:β	K:β

Table 3: SC dispatch table

In languages where a message can be sent to an object that does not understand it (i.e., dynamically typed languages), this approach is not quite as efficient as STI. In STI, a message is not understood only if the entry in the table for the class/selector pair is not associated with a meaningful method address. Recall that in this case it is initialized with the address of a function that reports an appropriate error message. However, in the colored table, two or more selectors can share the same row, so the wrong message may be invoked.

As an example, suppose that a message is sent to an instance of class F with selector ν. Since selector ν shares color 1 with δ the address in the table is F:δ, from Table 3. However, from Figure 1, class F does not understand selector ν, and so the dispatch technique must somehow detect this.

It is common to add a *method prologue* at the beginning of every method definition, which tests the current selector (passed as a hidden argument in every method invocation) against the expected selector (which is known at compile-time). If the comparison fails, an apropriate error message is generated. Otherwise, the rest of the method code is executed.

A non-incremental algorithm for selector coloring is presented in [DMSV89], and an incremental version in [AR92]. Some terminology is necessary:

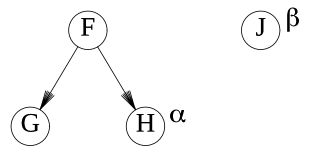
1. *Conflict Table*: each row, r in a conflict table represents a particular selector, $r.\sigma$, and stores the set of selectors, r.V, that *conflict* with σ. Two selectors conflict if any class in the environment understands both.
2. *Partition type*: Each class/selector pair $< C, \sigma >$ is assigned one of four different partition types:
   (a) *specific*: σ is not yet defined in the system
   (b) *separate*: σ is not recognized by class C [2], any superclass of C, or any subclass of C, but is recognized by some class (i.e. is not specific).
   (c) *declared*: σ is not recognized by class C or any superclass of C (and is not specific or separate)
   (d) *redefined*: σ is recognized by C.
3. *colorsFreeFor(G)*: The set of all colors unused by all classes in the set G. A class is using a color,L, if it recognizes a selector whose color is L.
4. *classesUsingColor(L)*: The set of classes using color L.

In [DMSV89], the non-incremental algorithm for selector coloring is divided into two parts: conflict table calculation, and color assignment.

```
Algorithm SC-static
    "compute conflict table"
    foreach selector $\sigma$
        R := conflict table row for $\sigma$
        foreach selector $\sigma_i$
            if $\exists C$ that recognizes $\sigma_i$
                add $\sigma_i$ to R.V
    "assign colors"
    foreach row R in conflict table
        index($\sigma$) := smallest index not in R.V
end SC-static
```

In [AR92], an incremental version of SC is presented. However, the declarative nature of the presentation does not provide any indication of how to implement the algorithm efficienctly. Furthermore, some errors exist in the algorithm. We present a procedural version of the [AR92] algorithm, and discuss it.

[2][AR92] does not explicitly exclude class C. It should be excluded so that after σ is added to C, the partition type of $< C, \sigma >$ becomes *redefined*



index	F	G	H	J
0	-	-	H:α	J:β

Figure 2: Example dispatch table

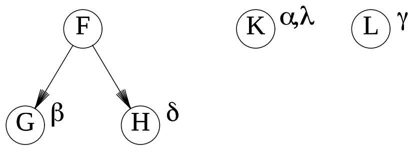
```
Algorithm SC
    K := -1
    foreach class $C$
        K := K+1
        index(C) := K
        foreach selector $\sigma$
            $L_{old}$ := index($\sigma$)
            P := partition($\sigma$,C)
            if P = specific
                L := any color in colorsFreeFor(subclasses(C))
            elsif P = redefined
                L := $L_{old}$
            elsif P = separate
                if $L_{old} \in$ colorsFreeFor($C$) then L := $L_{old}$
                else L := any color in colorsFreeFor(classesUsingColor($L_{old}$))
            else "P = declared"
                if $L_{old} \in$ colorsFreeFor($C$) then L := $L_{old}$
                else L := any color in colorsFreeFor(classesUsingColor($L_{old}$))
            index($\sigma$) := L
            T[L,K] := methodFor($\sigma$,C)
end SC
```

There are a few errors in the AR algorithm. If $< C, \sigma >$ is specific, [AR92] states that the color for σ can be any color free for all subclasses of C. However, if we assume that inheritance exceptions are represented as special method definitions (i.e. a method still exists for the selector, but just generates an error), then it is sufficent to check only the leaf classes of C. If inheritance exceptions do actually remove the selector, then class C and all subclasses must be checked.

If $< C, \sigma >$ is separate, it is not sufficient to check only class C to determine if the color can remain unchanged. Subclasses of C must also be checked. Once again, however, if inheritance exceptions are modeled as special methods, only leaf classes need to be checked.

As an example illustrating why AR is not sufficient, suppose we have the color map and inheritance graph of Figure 2. If we add a method for β to class F, then although color 1 is free for class F, it cannot be used for β since selectors α and β in class H cannot share the same color.

If $< C, \sigma >$ is partition type *declared*, the [AR92] specification is in err on two counts. First, it is not sufficient to look only at classes using the current



	F	G	H	K	L
1	-	$G.\beta$	-	$K.\alpha$	$L.\gamma$
2	-	-	-	$K.\lambda$	-
3	-	-	$H.\delta$	-	-

Figure 3: The color map for a simple inheritance graph

color unless a deletion mechanism is used to collapse rows. Second, the AR algorithm is too restrictive. That is, it may exclude a color that can be used. Instead of finding a color free for classes using the current color, the algorithm should find a color free for all dependent classes of $< C, \sigma >$ and free for all classes currently using selector σ. Dependent classes will be discussed later.

Consider Figure 3 where we are trying to add the selector γ to class F. The AR algorithm seeks a color free for the class set, G, K, L. Therefore, color 3 appears acceptable. However, color 3 cannot be used since G, a dependent class of $< F, \sigma >$ is already using color 3 for β (since class G will inherit γ it cannot use color 3 for both β and γ). Observe that the way in which this example was constructed, rows 1 and 3 can be collapsed into a single row, in which case the formula would work properly. In general, any example which demonstrates the above problem will have this property that rows could be collasped. Thus, this first problem with the AR formula is avoided only if the coloring algorithm includes a deletion mechanism that ensures that rows are fully collapsed. However, the time complexity of detecting such potential collapses may not be justified, in which case the AR formula is insufficient.

Consider Figure 3 again. The AR algorithm excludes color 2 since class K uses it (for selector λ). However, color 2 can be used for β since class K does not recognize selector γ and is not a dependent class of F.

2.3 Row Displacement (RD)

Row Displacement ([DH95]) compresses the two-dimensional STI table into a one-dimensional master array. Selectors are assigned unique indices in such a way that when all selector rows are shifted to the right by the index amount, the two-dimensional table has only one method in each column. The table is then collasped into a one-dimensional array. When dispatching a method invocation, the shift index of the selector and the index of the receiver class are added together to determine the index of the desired address within the master array. It is also possible to shift classes instead of selectors, as shown in [Dri93b]. However, it is observed in [DH95] that shifting selectors yields better compression rates. Table 4 shows how the class/selector table of Table 2 can be compressed using this technique:

Figure 4: RD dispatch table

It is important that the shift indices be unique, to ensure that two different selectors will not access the same dispatch table location for the same class. For example, no address conflicts occur if row 3 (selector ν) is assigned a shift index of 0 (M:ν can fit into the fifth slot of the array without conflicting with any other selector). However, this is not allowed since row 1 (selector δ) has already claimed shift index 0. If ν was assigned shift index 0, a dispatch on $< F, \nu >$ would invoke F:δ, instead of detecting a non-understood method. Although this situation could be detected by extending the method prologue to compare both the selector and class against their expected counterparts, it is more efficient to enforce uniqueness of selector shift indices so as not to incur an extra comparision on every dispatch. Similarily, row 4 (selector α) can not have shift index 2 because it would conflict with row 2 (selector β), even though this results in leaving the fifth slot empty.

In order to present an algorithm computing an RD dispatch table, we need the following terminology:

1. *Table*: the table, T, is a one-dimensional *master array*. A selector index, L, and class index, K, identify the entry T[K+L].

2. *Block*: a block is a structure representing a contiguous collection of class indices. It contains a starting index, *start*, and a block length, *run*.

3. *Row*: a row structure contains a selector, σ, and a collection of Blocks representing all classes which use σ. The number of such classes is referred to as the *width* of the row. The *primary block* of a row is the block with the largest run.

4. *Free(s)*: The entries in the table T can be divided into two categories, used and unused. All unused entries can be described by Blocks. That is, if entry T[i-1] is used, and entry T[i+r] is the next used entry, a *free block* with start i and run r can be used to represent all unused entries between these two entries. *Free(s)* is a doubly linked list of all free blocks whose

size is *s*. *firstFree(s)* returns the smallest free block (across all freelists) whose size is greater-equal *s*. *nextFree(F)* returns the freeblock after F, unless F doesn't have any more freeblocks, in which case it returns the result of calling firstFree(F.run+1).

5. *DRO sort order*: The row structures are to be sorted in descending order based on row width. All rows with width 1 are to be sorted in descending order based on the start index of their primary block.

```
Algorithm RD
    assign class indices in depth first preorder
    create a Row structure for each selector $\sigma$
    perform a DRO sort on the collection of Row structures
    foreach row R with width > 1 (in DRO order)
        L := unassigned
        F := firstFree(R.primary.run)
        while L is unassigned
            max := F.run - R.primary.run
            i := 0
            while L unassigned and $i \leq max$ do
                L := F.start - R.primary.start + i
                foreach non-primary block B in R
                    for K := B.start to B.start + B.run
                        if T[L+K] is used
                            L := unassigned
                            break two levels
                i := i+1
            if L unassigned
                F := nextFree(F)
        foreach block B in R
            F := the freeblock containing entry T[L,B.start]
            for K := B.start to B.start + B.run
                T[L,K] := methodFor(R.$\sigma$, classWithIndex(K))
            update free lists (split F into two smaller freeblocks)

    form a singly linked list of every free entry in the master array
    F := firstFree(1)
    foreach row R with width = 1
        L := F.start - R.primary.start
        T[L,K] := methodFor(R.$\sigma$, classWithIndex(K))
        F := F.next
end RD
```

There are only two real differences between the incremental version of RD dispatch provided by the DT algorithms and the non-incremental version provided in [DH95]. The first difference has to do with the optimizations the non-incremental version can make because it has access to the entire class hierarchy before selector index assignment begins. The non-incremental version sorts selectors according to how many classes recognize them; such sorting is not possible in an incremental algorithm. The non-incremental version relies on this sorting to fit all selectors with a width of one last. During the fitting of selectors with width greater than one, the algorithm does not worry about maintaining a single-entry freelist (with only one entry, the doubly linked freelist structure could not be encoded into the master array directly). In this way, the algorithm can ignore single-entry freeblocks until the width 1 selectors are encountered, at which time a pass is made through the main array to generate a singly linked list of 1 element freeblocks. This pass destroys the doubly linked list of arbitrarily sized freeblocks maintained before, but in the non-incremental version such freeblocks are no longer needed by this time.

The incremental version cannot sort selectors by width, and cannot rely on one-entry selectors occuring last. Thus, doubly linked single-entry freeblocks must be maintained like any other size of freeblock. Fortunately, tables in the DT Framework store divisions rather than method addresses, so a special FreeDivision can be used to represent freeblocks. Thus, even single-entry freeblocks can encode the doubly-linked freeblock structure within the master array (FreeDivision instances have *next* and *previous* fields pointing to other FreeDivision instances representing freeblocks of the same size.

[DH95] discusses the impact of selector ordering on both execution performance and fillrate. It can be expected that execution performance and fillrate will degrade in the incremental version. This is discussed in Section 7.

2.4 Compact Selector-Indexed Dispatch Tables (CT)

Compact Selector-Indexed Dispatch Tables ([VH96]) compress the STI table by using four different strategies: *selector separation*, *selector aliasing*, *class partitioning*, and *class sharing*. Selector separation divides selectors into two groups: *standard selectors* have one main definition and are only overridden in subclasses, and any selector that is not standard is a *conflict selector*. Two different tables are maintained, one for standard selectors, the other for conflict selectors. Selector aliasing can be performed only on the standard selector table, and relies entirely on classes being sorted top-down and having at most one parent class. Thus, CT dispatch as presented in [VH96] is limited to single inheritance languages.

```
Algorithm CT
    Order classes top-down
    Separate selectors into standard and conflict sets

    “Standard Table Index Assignment”
    K := -1
    foreach class C (ordered top-down)
        L := -1
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            L := L+1
            index(S) := L
            T[L][K] := methodFor($\sigma$,C)
        “Conflict Table Index Assignment”
    L := -1; K := -1
    foreach class C
```

```
        K := K+1
        index(C) := K
        foreach selector $\sigma$ recognized by C
            if index($\sigma$) is unassigned
                L := L+1
                index(S) := L
            T[L][K] := methodFor($\sigma$,C)

    Partition standard table into subarrays, each with $p_s$ elements
    Partition conflict table into subarrays, each with $p_c$ elements

    Within each partitioned subtable, merge identical columns together
end CT
```

The CT technique obtains its excellent compression from two distinct mechanisms. First, by relying on single inheritance and knowledge of all classes in the environment, selector indices in the standard table are assigned on a per-class basis. However, this never results in an index being assigned different indices in different classes as long as the order in which selectors are traversed remains constant across classes. The result of this is that all internal space in the STI table for standard selectors is entirely removed (that is, the only unused space is at the end of each column). The separation of selectors into standard and conflicting provides this selector aliasing capability.

Second, class sharing can substantially reduce the amount of space taken up by the table, especially for small partition sizes, p_s and p_c. However, a reduction in table size does not necessarily imply a reduction in overall memory utilization, because there is memory overhead involved in maintaining partitions, as discussed in [VH96]. Without partitioning, class sharing will almost never provide any benefit, but with judicious choices for partition sizes, this technique uses less space than any other.

An incremental version of the CT dispatch technique as it exists in [VH96] necessitates some inefficiency, due to the inherently non-incremental nature of selector aliasing. In an incremental version, classes can be added as parent classes of already existing classes. Since selector aliasing relies on assigning selector indices based on a top-down traversal of classes, this would result in a need to change the indices of many selectors. Although the index reassignment itself is not particularily expensive, the movement of divisions from old locations to new locations can involve a reshuffling of the entire table.

Fortunately, a simple observation makes incremental selector aliasing unnecessary; the standard table can be compressed equally well by using selector coloring. Having separated conflict selectors out of the table, selector coloring will assign indices so as to not leave any internal space (however, later we will discuss certain optimizations for SC dispatch that will result in a few internal spaces, in exchange for faster performance).

Having resolved the issue of incremental selector aliasing, we now turn our attention to incremental class partitioning and class sharing. Rather than creating standard and conflict tables in their entirety, then partitioning them, we can maintain fixed-size subtables that represent each partition. As addresses are added to the table, new subtables can be dynamically created as they are needed. Although an extremely efficient mechanism for incremental type sharing exists as long as we disallow adding of parent classes to existing classes, it is even possible (albeit more inefficient) to handle dynamic schema evolution.

Thus, the incremental version of CT consists of a table with two subtables, a standard selector table and a conflict selector table. Selectors exist in only one or the other of these tables, but the same class can exist in both (thus, class indices are selector dependent). Furthermore, each of these two subtables is divided into a collection of fixed-row subsubtables representing partitions. Each subsubtable in the standard selector subtable is compressed via selector aliasing and class sharing, and each subsubtable in the conflict selector subtable is compressed via class sharing alone.

As will be discussed in some detail later, the incremental version of CT is only one of many variations arising from separated and partitioned tables. Later, we will introduce a new dispatch technique, SCCT, that merges the SC and CT dispatch techniques, keeping the advantages of both, and removing the limitations of CT. In particular, SCCT is applicable to languages with multiple inheritance, and provides even better compression than CT.

2.5 Virtual Function Tables (VTBL)

Virtual Function Tables ([ES90]) have a different dispatch table for each class, so selector indices are class-specific, although they are constrained to be equal across inheritance subgraphs. Since this constraint is not possible in multiple inheritance, each class stores multiple tables; for selector σ, class C has as many tables are there are classes in selectorRoot(σ) that are superclasses of C.

1. *Inheritance Paths*: An inheritance path for the class/selector pair $< C, \sigma >$ is defined as an ordered collection of classes $C_1, C_2, ..., C_k$ in which $C_1 \in$ parents(C), $C_i \in$ parents(C_{i-1}), and $C_k \in$ rootClasses(σ). Multiple paths are induced by multiple inheritance

```
Algorithm VTBL
    foreach selector $\sigma$
        foreach class C (sorted top-down)
            if $\sigma \in$ selectors(C)
                V := C.vtbl[0]
                L := V.size
                V[L] := methodFor($\sigma$,C)
                index($\sigma$,C) := L
            else
                foreach inheritance path $P_i$ for $< C, \sigma >$
                    if $\exists C_k$ in $P_i$
                        V := C.vtbl[i]
                        L := index($\sigma$,C)
                        V[L] := methodFor($\sigma$,C)
end VTBL
```

Unfortunately, an incremental version of the VTBL technique is expensive for two reasons. First, it is not possible to store all current selector indices explicitly, because selector indices are class specific. This problem exists for the same reason STI dispatch is not practical; the product of classes and selectors requires far more memory than is feasible. This means that selector index determination becomes a search, rather than just a field access. Even efficient implementations like hash tables with binary search tree probes will be an order of magnitude more expensive than selector index determination in any other technique.

The second inefficiency is due to the need to handle dynamic schema evolution. If a class is added as a parent of an existing class, C, all selectors defined in C or any subclass of C which are not defined in any parent of C must have their indices reassigned. Thus, if a class is added as a parent of a hierarchy with a single current root class, every selector of every class in the hierarchy must be assigned a new index.

Note that although an incremental VTBL technique is potentially very expensive, it is not impossible. It could even be used in reflexive languages, as long as every virtual function table used thunks, rather than just those tables involving multiple inheritance. However, since this would have a profound impact on execution performance, we have not included VTBL dispatch in the DT Framework.

3 The DT Algorithms

The DT algorithms interact with a few fundamental data structures in order to modify dispatch table information incrementally when the programming environment changes. The environment changes (from the perspective of the DT algorithms) when selectors or class hierarchy links are added or removed. We will refer to these four actions as *environment modifications*. These actions are divided into two categories: *method adding* occurs when selectors and class links are added, and *method removal* occurs when selectors and class links are removed. Data structures to represent classes and selectors are needed. Classes maintain a name, a set of native selectors, a set of parent classes, and a set of child classes. Selectors maintain only a name. The algorithms also need data structures to represent two special constructs, *divisions* and *division tables*. These are discussed in subsections that follow.

There are four DT algorithms that act as the interface to the framework. They correspond to the four fundamental operations that cause environment modification: adding a selector (Algorithm AS), removing a selector (Algorithm RS), adding class hierarchy links (Algorithm ACL) and removing class hierarchy links (Algorithm RCL). Note that defining a class does not itself modify the dispatch information. Only when selectors are added, or the class is connected to other classes via inheritance, does the dispatch information change. In addition to the interface algorithms, there are some fundamental algorithms to perform inheritance management, inheritance conflict detection, index determination, and index conflict resolution. The DT algorithms, and their overall purposes, are summarized in Table 4. Each algorithm is presented in subsections after we introduce divisions and division tables.

Algorithm	Semantic Name	Algorithm Purpose
AS	Add Selector	Add a selector to an existing class
RS	Remove Selector	Remove a selector from an existing class
ACL	Add Class Links	Add inheritance links to a class
RCL	Remove Class Links	Remove inheritance links from a class
IMA	Inheritance Manager, Adding	Inheritance propogation and conflict detection
IMR	Inheritance Manager, Removing	Inheritance propogation and conflict detection
DSI	Determine Selector Index	Assign an index to a selector
DCI	Determine Class Index	Assign an index to a class

Table 4: DT Algorithm Purposes

3.1 Divisions and the Generalized Dispatch Table

One of the most fundamental DT Framework concepts is that of a *division*. Divisions are the mechanism by which both the functionality and efficiency of the DT algorithms is provided. A division represents a method to be executed for a particular class/selector pair, but contains extra information in addition to the method address (usually, a class and a selector). The class associated with a division is referred to as the *defining class* of the division.

The Table class and its subclasses represent extended dispatch tables called *division tables*, which store division pointers instead of addressed. By storing divisions in the tables, rather than simple addresses, the following capabilities become possible:

1. Localized modification of the division table during environment modification so that only those entries that need to be recomputed are affected.

2. Efficient inheritance propogation and inheritance conflict detection.

3. Detection of simple recompilations (replacing a method for a selector by a different method) and avoidance of unnecessary computation in such situations.

4. compile-time method determination.

Every entry of a division table contains a division instance, including entries that do not have user-specified methods associated with them. Such empty entries usually contain a special unique *EmptyDivision* instance, but some indexing strategies use *FreeDivision* instances, which represent a contiguous block of unused table entries. Instances of both of these classes have a special *methodNotUnderstood* address associated with them.

Table entries corresponding to class/selector pairs having a user-specified method are *standard divisions*, and as such have a *defining class*, *selector*, *address* and a set of child divisions. Two alternative forms of standard division exist, normal divisions and conflict divisions. A *normal division* is used to represent a user-provided method for a particular class/selector pair. A *conflict division* is used to represent inheritance conflicts due to multiple inheritance. If a class C inherits two or more distinct methods for a selector σ, a conflict division is created with defining class C, selector σ and a special *inheritance-Conflict* method address. Note that this mechanism for handling inheritance conflicts implies that conflicts in a class C result in an implicit native definition of selector σ in class C.

Associated with standard divisions is the concept of dependent classes. For a division D representing class/selector pair $< C, \sigma >$, the *dependent classes* of D consist of all classes which inherit selector σ from class C. Furthermore, each selector σ defined in the environment generates a *division inheritance graph*, which is an induced subgraph of the class inheritance hierarchy, formed by removing all classes which do not natively define σ. Division hierarchy graphs are what allow division tables to perform compile-time method determination. These graphs can be maintained by having each division store a set of child divisions. For a division D with defining class C and selector σ, the child divisions of D are the divisions for selector σ and classes C_i immediately below C in the division inheritance graph for σ.

3.2 The DT Algorithms

3.2.1 Algorithms AS and RS (Add/Remove Selector)

Algorithm AS is one of the interface routines provided by the DT Environment. Each time a compiler encounters a new method declaration for a selector, σ, in a particular class, C, it calls this routine (the compiler is assumed to have made an instance of the DT Environment before it started any parsing). As well, a run-time system that encounters a method declaration at run-time does exactly the same thing, calling Algorithm AS with the appropriate selector and class arguments.

```
Algorithm AS(inout $\sigma$ : Selector, inout C : Class, in A : Address, inout T: DivisionTable)

1   if index($\sigma$) = unassigned or ( T[$\sigma, C$] $\neq \Omega$ and T[$\sigma, C$].$\sigma \neq \sigma$ ) then
2       DSI($\sigma, C, T$)
3   endif

4   $D_C$ := T[$\sigma, C$]
5   if $D_C.C$ = C and $D_C.\sigma = \sigma$ then
6       $D_C.A$ := A
7       remove any conflict marking on $D_C$

8   else
9       insert $\sigma$ into selectors(C)
10      $D_N$ := newDivision($C, \sigma, A$)
11      addDivisionChild($D_C, D_N$)
12      IMA($C, C, D_N, nil, T$)
```

```
13    endif
end AS
```

Lines 1-3 of Algorithm AS determines whether a new selector index is needed, and if so, calls Algorithm DSI (Determine Selector Index) to establish a new index and move the division as appropriate.

Lines 4-7 determine whether a *method recompilation* or *inheritance conflict removal* has occured. In either case, a division already exists that has been propogated to a dependent class, so no propogation is necessary. Since the table entries for all dependent classes of $< C, \sigma >$ store a pointer to the same division, assigning the new address to the current division has the effect of modifying the information in multiple division table entries simulaneously.

If the test in line 5 fails, Algorithm AS falls into its most common scenario, lines 8-12. A new division is created, a division hierarchy link is added, and Algorithm IMA is called to propogate the new division to the child classes.

Algorithm RS makes all necessary adjustments to the division table and related data structures when a selector is removed from a particular class. It is impossible for selector index conflicts to occur when removing a class. On the other hand, it is possible for inheritance conflicts to occur, when the native selector definition being removed hides an otherwise multiply visible selector.

```
Algorithm RS(in $\sigma$ : Selector, in C : Class, in T : DivisionTable)

1     remove $\sigma$ from selectors(C)

2     $D_N := \text{ID}(\sigma, C, parents(C), \{\}, T)$

3     $\text{IMR}(C, \sigma, D_N, nil, T)$

end RS
```

3.2.2 Algorithm IMA/IMR (Inheritance Management)

Algorithm IMA, and its interactions with Algorithms AS and ACL, form the most important parts of the DT algorithms (along with the analogous case for Algorithms IMR, RS and RCL). Algorithm IMA is responsible for propogating a division provided to it from Algorithm AS or ACL, to all dependent classes of the division. During this propogation the algorithm is also responsible for maintaining inheritance conflict information and managing selector index conflicts.

Algorithm IMA is a recursive algorithm that is applied to one class, then to each child class of that class. Recursion terminates when a class with a native definition is encountered, or no child classes exist. The algorithm has five arguments, but two of them are critical: the class on which the current recursive invocation applies, and the division to be propogated. The class is referred to as the *target class*, and denoted by C_T. The division is referred to as the *new division*, and denoted by D_N. The other arguments will be discussed later. For now, simply note that each invocation of the algorithm is attempting to propogate a new division, D_N to a particular target class, C_T.

Notation	Definition
D_C	The current division, T[σ, C_T]
D_N	The new division (established in Algorithm AS, ACL, RS or RCL)
C_T	The current target class, on which IMA/IMR is currently invoked.
C_N	The defining class of the new division. Shorthand for $D_N.C$.
C_I	The class from which C_T currently inherits the method for $D_N.\sigma$
C_B	The class from which division propogation is to begin
π	Boolean test indicating whether, after D_N has been added to the division table, $D_N.\sigma$ is visible in C_T from both C_N and C_I, where $C_N \neq C_I$.

Table 5: Notation and Definitions for IM Algorithms

Within a particular invocation of Algorithm IMA, the primary goal is determining which division should be placed in the division table for $< C_T, D_N.\sigma >$. There are only three possibilities: 1) the new division, D_N is inserted into the table, 2) the division, D_C, that currently exists in the table for the entry is left untouched, or 3) a new division is created/obtained to be placed in the table.

These three possibilities correspond to three distinct scenarios. In the discussion of these scenarios, σ refers to $D_N.\sigma$. Also, note that in Algorithm IMR, method removal actually refers to the propogation of a division, since removal of a method is implemented by propogating (adding) an appropriate division.

1a *Division inserting (DI)*: This scenario occurs when we have previously established that the new division, D_N, should be placed in the table for all dependent classes of $< C_B, \sigma >$. Thus, scenario DI occurs when C_T is a dependent class of D_N, and consists solely of inserting D_N into the division table and continuing recursion.

1b *Division re-inserting (DRI)*: In class hierarchies with multiple inheritance, there is often more than one path from a base class, C_B to an arbitrary subclass, C_T. This implies that during a recursive traversal of child classes, our inheritance management algorithm can visit the same target class more than once. However, on the second and subsequent visits, absolutely no work needs to be done. Scenario DRI occurs when $D_N = D_C \neq \Omega$ and consists solely of terminating the recursion.

2 *Division child updating (DCU)*: Termination of the recursive traversal of the class hierarchy stops when a class is detected which has a native declaration for σ. In this case, we want to leave the current division, D_C, as is, since native definitions override inherited ones. However, since each division maintains the set of its child divisions, we must update these links. Scenario DCU occurs when a native definition (implicit or explicit) for σ exists in C_T, and involves updating division child information and stopping recursion.

3a *Conflict-creating (CC)*: In Algorithm IMA, propogating a division can result in an inheritance conflict. The boolean test π from Table 5 is useful because an inheritance conflict exists in C_T if the test is true, and does not exist in C_T if it is false. We will discuss how to efficiently determine the truth value of π later. Note that D_C represents the method that C_T currently executes for selector σ. Furthermore, $D_C.C$ represents the defining class of this method. Scenario CC occurs when there exists a path between C_T and $D_C.C$ which does not pass through $D_N.C$. It involves creating a conflict division and propogating this division to all dependent classes of $< C_T, D.\sigma >$.

3b *Conflict-removing (CR)*: In Algorithm IMR, propogating a division can result in the removal of an existing inheritance conflict. Scenario CR occurs when D_C is a conflict, there exists exactly two parent divisions of D_C (i.e. $\mid D_C.P \mid = 2$), and either D_N is empty or is an element of $D_C.P$. It involves propogating the single division element of $D_C.P - \{D_N, D_R\}$ to all dependent classes of $< C_T, D.\sigma >$, where D_R refers to the division being removed.

Four fundamental boolean tests exist that allow us to efficiently determine what scenario should be performed during a particular invocation of Algorithm IMA or IMR.

The four tests are:

1. $C_T = C_I$ (does a native definition exist?)
2. $C_N = C_I$ (have we already propogated a division to this class?)
3. $C_I = nil$ (does the current class recognize the selector in question?)
4. $\pi =$ true (after adding D_N, does an inheritance conflict exist?)

Table 6 shows how these four tests efficiently determine which scenario to perform during Algorithms IMA and IMR. Many combinations of truth values are not possible because the four tests are not entirely independent. Each combination of truth values that is not possible has one or more assertion numbers, presented in Appendix A, explaining why it is not possible. Legal truth value combinations are marked with the appropriate scenario to perform.

From Table 6, we can determine which tests are necessary to identify the desired scenario during an invocation of Algorithms IMA and IMR. Tests for Algorithm IMA are summarized in Table 7 and tests for Algorithm IMR are summarized in Table 8.

All of these tests are simple comparisons, except for determining the truth value of π. Remember that π is true if σ is visible in C_T from both C_N and C_I, when $C_N \neq C_I$. It is useful because an inheritance conflict exists in C_T if the test is true, and does not exist in C_T if it is false. A naive algorithm could determine the truth value of π by traversing down the inheritance hierarchy from both C_N and C_I, looking for C_T. However, a much more efficient mechanism exists.

$C_T = C_I$	$C_N = C_I$	$C_I = nil$	π	IMA scenarios	IMR scenarios
T	T	T	T	1,4,8,10 or 11	1,6,8,10 or 11
T	T	T	F	1,4 or 8	1,6 or 8
T	T	F	T	5,8 or 11	5,8 or 11
T	T	F	F	8	8
T	F	T	T	1 or 10	1,6 or 10
T	F	T	F	1 or 10	1 or 6
T	F	F	T	**DCU**	**DCU**
T	F	F	F	**DCU**	if isConflict(D_C) **CR** else **DCU**
F	T	T	T	4,10 or 11	6, 10, or 11
F	T	T	F	4	6
F	T	F	T	11	11
F	T	F	F	**DRI**	**DRI**
F	F	T	T	10	6 or 10
F	F	T	F	**DI**	6
F	F	F	T	**CC**	12
F	F	F	F	**DI**	**DI**

Table 6: All truth combinations of the four fundamental DT tests

Scenario	Tests
DCU	$C_T = C_I$
DRI	$C_T \neq C_I$ and $C_N = C_I$
CC	$C_T \neq C_I$ and $C_N \neq C_I$ and π = true
DI	$C_T \neq C_I$ and $C_N \neq C_I$ and π = false

Table 7: Determining scenario during IMA invocations

Even though the truth value of π asssumes that D_N has already been added, it is possible to use information stored in the table before D_N is placed to efficiently determine π. In Algorithm IMA, we define $\Sigma = \{D \mid D = T[D_N.\sigma, C_i], C_i \in parents(C_T)\} - \{\Omega\}$. That is, Σ represents the set of non-empty divisions stored in the division table for all parent classes of C_T. If $\Sigma > 1$, a conflict would exist if D_N were added to C_T. When C_T has a native definition for σ, Σ is identical to $D_C.P$, where D_C is the division T$[\sigma, C_T]$, and $D.P$ is the set of parent divisions of D.

For Algorithm IMR, Σ is defined as for Algorithm IMA, except that the division being removed, D_R, is not considered as part of the set. Later, we will see that in Algorithm IMR, D_N does not refer to D_R, but rather to the division that should be visible in C_T if D_R were removed. This necessitates some other mechanism for obtaining D_R, which will be discussed when Algorithm IMR is presented. In any event, once Σ has been obtained, if $\mid \Sigma \mid > 1$, a conflict would exist in C_T if D_N were added (i.e. if D_R were removed).

There are also certain times when computation of Σ is not even necessary. First, π is immediately true if $C_I < C_N$ (from Assertion 9 of Appendix A). Second, π can never be true if C_T has only one parent class (σ cannot be multiply visible if there is only one path by which selectors can be visible). Third, π can never be true if $C_N = C_I$ (from the definition of π). Thus, an

Scenario	Tests
DRI	$C_T \neq C_I$ and $C_N = C_I$
DI	$C_T \neq C_I$ and $C_N \neq C_I$
CR	$C_T = C_I$ and isConflict(D_C) and π = false
DCU	$C_T = C_I$ and (not isConflict(D_C) or π = true)

Table 8: Determining scenario during IMR invocation

efficient test for establishing the true value of π is: ($C_I < C_N$) or ($C_N \neq C_I$ and $\mid parents(C_T) \mid > 1$ and $\mid \Sigma \mid > 1$).

It is possible for this test to generate temporary conflicts where they do not truly exist, during a particular invocation. However, by the time all invocations of Algorithm IMA/IMR are finished (for a particular invocation of Algorithm AS,RS,ACL or RCL), such temporary conflicts will be removed.

So far, we have determined the possible scenarios that can occur during inheritance propogation, and found efficient tests for establishing which scenario is applicable during a particular invocation of Algorithms IMA and IMR. However, before presenting the algorithms, there is an important issue that must be discussed. Up to this point, we have not explained in any detail the role that a selector index plays in the division tables. We mentioned previously that the selector index establishes a starting location within the table, and that the exact interpretation of the index depends on the dispatch technique used. We must discuss this in more detail, because Algorithm IMA needs to be aware of a special type of conflict called a selector index conflict. A *selector index conflict* can occur in certain table-based dispatch techniques because selector indices are not necessarily unique. Two different selectors can share the same index as long as only one non-empty division needs to be stored in a particular division table entry at a given time. A selector index conflict occurs when an attempt is made to insert a division into a division table entry that already contains a non-empty division with a different selector. In these situations, one of the selectors must be assigned a new index, and all divisions in the division table associated with that selector must be moved to new locations, based on the new index value.

Algorithm DSI (Determine Selector Index) is responsible for assigning a legal index to a selector. It is presented in Section 3.2.4. Algorithm DSI needs to be invoked in two distinct situations: 1) when the current selector does not yet have an index (i.e. its index is *unassigned*), and 2) when a selector index conflict is detected. Algorithm AS only needs to invoke Algorithm DSI when the index, L, of the current selector, $D_N.\sigma$, is unassigned. Otherwise, Algorithm AS assumes that no selector index exists and calls Algorithm IMA. Algorithm IMA is perfectly suited for detecting selector index conflicts, and it directly invokes Algorithm DSI when it detects a conflict. Detecting a conflict involves a simple test: $D_C \neq \Omega$ and $D_C.\sigma \neq D_N.\sigma$. If this test is true, a selector index conflict exists, and Algorithm DSI is called to obtain a new selector index for $D_N.\sigma$ and move all existing divisions for $D_N.\sigma$ to the new table entries indicated by this new index.

Note that Algorithm DSI can be called during any recursive invocation of Algorithm IMA even though this means that, at the time it is called, the new division has only been propgated to some of the dependent classes. Algorithm DSI will move the already propogated divisions to their new locations, and the subsequent recursive invocations will have a new selector index, L, thus placing divisions in their correct locations.

Unlike Algorithm IMA, Algorithm IMR does not need to worry about selector index conflicts, because it propogates either empty divisions or divisions that already exist in the table.

Having established the possible scenarios for a particular invocation of Algorithm IMA, as well as how to efficiently determine which scenario to perform, we are ready to present Algorithm IMA. It has five arguments:

1. C_T, the current target class.
2. C_B, the base class from which inheritance propogation should start (needed by Algorithm DSI)
3. D_N, the new division which is to be propogated to all dependent classes of $< C_B, \sigma >$.
4. D_P, the division in the table for the parent class of C_T from which this invocation occured.
5. T, the division table to be modified.

```
Algorithm IMA( in $C_T$ : Class, in $C_B$ : Class, in $D_N$ : Division,
    in $D_P$ : Division, inout T : Table)

    "Assign important variables"
    $\sigma := D_N.\sigma$
    $C_N := D_N.C$
    $D_N :=$ T[$C_N$,$\sigma$]
    $C_I := D_C.C$

    "Check for selector index conflict"
    if $D_C \neq \Omega$ and $D_C.\sigma \neq D_N.\sigma$ then
        DSI($D_N.\sigma$,$C_B$,T)
        $D_C :=$ T[$\sigma$, $C_T$]
        $C_I := D_C.C$
    endif

    "Determine and perform appropriate scenarios"
    if $C_T = C_I$ then "scenario DCU"
        addDivisionChild($D_N$, $D_C$)
        removeDivisionChild($D_P$, $D_C$)
        return

    elsif ( $C_I = C_N$ ) "scenario DRI"
        return

    elsif ( $\pi$ = true ) then
```

```
17          D := RIC($\sigma, C_T, \{D_N, D_C\}$)

18    else "scenario DI"
19          $D := D_N$

20    endif

      "Insert division and propogate to children"
21    T[$\sigma, C_T$, :]= D
22    foreach $C_i \in$ children($C_T$) do
23          IMA($C_i, C_B, D, D_C, T$)
24    endfor

end IMA
```

Algorithm IMA can be divided into four distinct parts. Lines 1-4 determine the values of the test variables. Note that $D_C = \Omega$ when $D_N.\sigma$ is not currently visible in C_T. We define $\Omega.C$ = nil, so in such cases, C_I will be nil.

Lines 5-9 test for a selector index conflict, and, if one is detected, invoke Algorithm DSI and reassign test variables that change due to selector index modification. Recall that Algorithm DSI is responsible for assigning selector indices, establishing new indicies when selector index conflicts occur, and moving all selectors in a division table when selector indices change. Note that selector index conflicts are not possible in STI and VTBL dispatch techniques, so the DT Table classes used to implement these dispatch techniques provide an implementation of Algorithm IMA without lines 5-9. Furthermore, due to the manner in which Algorithm DSI assigns selector indices, it is not possible for more than one selector index conflict to occur during a single invocation of Algorithms AS and ACL, so if lines 6-8 are ever executed, subsequent recursive invocations can avoid the check for selector index conflicts by calling the version of Algorithm IMA without them.

Lines 10-22 apply the scenario determining tests to establish one of the three scenarios. Only one of the three scenarios is performed for each invocation of Algorithm IMA, but in all scenarios, one of two things must occur: 1) the scenario performs an immediate return, thus stopping recursion and not performing any additional code in the algorithm or 2) the scenario assigns a value to the special variable, D. If the algorithm reaches the fourth part, variable D is to represent the division that should be placed in the division table for C_T, and propogated to child classes of C_T. It is usually D_N, but during conflict-creation this is not the case. In line 11, procedure *addDivisionChild* adds its second argument as a child division of its first argument. in line 12, procedure *removeDivisionChild* removes its second argument as a child of its first argument. In both cases, if either argument is an empty division, no link is added.

When the DT Algorithms are used on a language with single inheritance, conflict detection is unnecessary and multiple paths to classes do not exist, so scenarios *conflict-creating* and *division re-inserting* are not possible. In such languages, Algorithm IMA simplifies to a single test: if $C_T = C_I$, perform *division child updating*, and if not, perform *division inserting*.

Finally, Lines 21-24 are only executed if the scenario determined in the third part does not request an explicit return. It consists of inserting division D into the division table for $< C_T, \sigma >$ and recursively invoking the algorithm on all child classes of C_T, passing in the division D as the division to be propogated. It is important that division table entries in parents be modified before those in children, in order for π to be efficiently determined.

The arguments to Algorithm IMR are similar, but not identical to those for Algorithm IMA. Selector index conflicts cannot occur in Algorithm IMR, and since C_B, the base class, is needed only for passing to Algorithm DSI, C_B is not necessary for Algorithm IMR. However, it is necessary to explicitly pass in the selector for which the removal is occuring, because the propogated division, D_N can be empty. In Algorithm IMA, this argument was not needed because it can be obtained from $D_N.\sigma$, since $D_N \neq \Omega$ (Assertion 4 of Appendix A).

```
Algorithm IMR( in $C_T$ : Class, in $\sigma$ : Selector, in $D_N$ : Division,
        in $D_P$ : Division, inout T : Table)
        "Assign important variables"
1       $\sigma := D_N.\sigma$
2       $C_N := D_N.C$
3       $D_N :=$ T[$C_N, \sigma$]
4       $C_I := D_C.C$
        "Determine and perform appropriate action"
5       if $C_T \neq C_I$ then
6           if $C_N = C_I$ then "action DRI"
7               return
8           else "action DI"
9               $D := D_N$
10          endif
11      elsif isConflict($D_C$) and not $\mid \Sigma \mid > 1$ then "action CR"
12          if $\mid \Sigma \mid = 0$ then
13              D := $\Omega$
14          else
15              D := the single element of $\Sigma$
16          endif
17      else "action DCU"
18          addDivisionChild($D_N, D_C$)
19          removeDivisionChild($D_P, D_C$)
20          return
21      endif
        "Insert division and propogate to children"
22      T[$\sigma, C_T$] := D
23      foreach $C_i \in children(C_T)$ do
24          $IMR(C_i, \sigma, D, D_C, G, T)$
25      endfor
end IMR
```

Algorithm IMR is divided into only three parts, since index conflicts are not possible. Lines 1-4 set the values of test variables. Note that for Algorithm IMR, C_I will never be nil because D_C will never be empty (it represents the division of the selector being removed, or a removed conflict division). However, since D_N can be empty, C_N can be nil. If this occurs, it indicates that no method for the selector is visible (in C_T) after the existing method is removed.

Lines 5-21 establish which scenario to execute, and perform the appropriate actions. In line 11, remember that we have established that the truth value of π, if D_N were added to C_T, is efficiently computable with the following test:

$(C_I < C_N)$ or ($C_N \neq C_I$ and $\mid parents(C_T) \mid > 1$ and $\mid \Sigma \mid > 1$). Everything in this test before Σ exists to avoid calculating Σ, but since Σ is needed in order to obtain a value for D, we must always compute it, so the other tests are not used. Recall that, for Algorithm IMR, Σ is the set of non-empty divisions stored for selector σ and all parent classes of C_T, where the division being removed is not considered part of the set. Since the division being removed is represented by D_P, we have all the information necessary to compute Σ. Also, notice from Table 8 that when $C_T = C_I$, it is not possible for $C_N = C_I$, so we can avoid that test. If π is false, there can be at most one element in Σ. Σ can also be empty, since it does not contain Ω — in such cases, D is assigned Ω. Otherwise, D is assigned the single element of Σ.

Lines 22-25 are only executed if the scenario determined in the second part did not perform an explicit return. The division table entry identified by $< C_T, \sigma >$ is modified, and the algorithm is recursively invoked on all child classes of class C_T.

3.2.3 Algorithms ACL and RCL (Add/Remove Class Links)

Algorithm ACL is responsible for updating the division table when new inheritance links are added to the inheritance graph. Dynamic schema evolution is possible, so new parent and child links can be added to a class which already has parent and/or child classes. Rather than having Algorithm ACL add one inheritance link at a time, we have generalized it so that an arbitrary number of both parent and child class links can be added. This is done because the number of calls to Algorithm IMA can often be reduced when multiple parents are given. For example, when a conflict occurs between one or more of the new parent classes, such conflicts can be detected in Algorithm ACL, allowing for a single conflict division to be propogated. If only a single parent were provided at a time, the first parent specified would propogate the division normally, but when the second (presumably conflicting) parent was added, a conflict division would have to be created and propogated instead. Algorithm ACL accepts a class C, a set of parent classes, G_P, and a set of children classes G_C.

```
Algorithm ACL(in C : Class, in $G_P$ : Set , in $G_C$ : Set, inout T : DivisionTable) : Boolean

1    update parent and child sets of all classes in $\{C\} \cup G_C \cup G_P$ as appropriate
2    if inheritance graph is cylic then
3         undo changes
4         return false
5    endif

6    if ( $\mid G_C \mid > 0$ ) then
7         foreach $\sigma \in$ selectors(C) do
8              D := T[$\sigma, C$]
9              foreach $C_i \in G_C$ do
10                  IMA($C_i, C, D, D, T$)
11             endfor
12        endfor
13   endif

14   if ( $\mid G_P \mid > 0$ ) then
15        G := ICB($C, G_P, T$)
16        for $< \sigma, D > \in G$ do
```

```
17              if not isEmpty(D) then
18                  IMA($C, C, D, nil, T$)
19              endif
20          endfor
21      endif

end ACL
```

Lines 1-5 are responsible for updating class hierarchy links and ensuring the inheritance graph remains acylic. Lines 7-12 propogate the native behavior of class C to classes in G_C. Note that it is neither possible, nor desirable, to invoke Algorithm IMA on class C directly. It is not possible, because this would result in $C_N = C_I = C_T$ within Algorithm IMA, which has been intentionally disallowed for efficiency reason. It is undesirable because it would result in division propogation to children that have already had propogation performed (since G_C need not be the entire set of child classes of C). Thus, we call Algorithm IMA in each child class found in G_C. In lines 15-20, Algorithm ICB (Inherited Class Behavior) returns the set of all divisions inerited in class C for σ from parents classes in the class set G_P. If different methods for the same selector are inherited, Algorithm ICB detects this and replaces the multiple divisions with a single conflict division to be propogated. Thus, the set G is guaranteed to have at most one division for each selector in the environment. All such divisions are propogated to class C and dependent classes of C by calling Algorithm IMA on C itself.

Algorithm RCL is used to update the division table when inheritance links between classes are removed.

```
Algorithm RCL(in C : Class, in $G_P$ : Set of Classes, in $G_C$ : Set of Classes, in T : DivisionTable)

1       remove classes in $G_P$ from parent set of C
2       remove classes in $G_C$ from child set of C

3       if ( | $G_C$ |> 0 ) then
4           foreach $\sigma \in$ selectors(C) do
5               foreach $C_i element G_C$ do
6                   $D_N$ := ID($\sigma, C_i, parents(C_i) - \{C\}, \{\}, T$)
7                   IMR($C_i, C, D_N, nil, T$)
8               endfor
9           endfor
10      endif

11      if ( | $G_P$ |> 0 ) then
12          G := ICB($C, parents(C) - G_P, T$)
13          for $< \sigma, D > \in G$ do
14              IMR($C, \sigma, D, nil, T$)
15          endfor
16      endif

end RCL
```

In line 5, similar to Algorithm ACL, we treat native selectors separately from inherited behavior. We iterate over every native selector in class C, and for each child class of C, obtain the appropriate division inherited in the child class, given that the child no longer inherits from C. Algorithm ID returns the division inherited in class C for selector σ if no native definition existed in C and C had as parents only the classes in the provided set.

In line 12, the inherited behavior consists of the behavior inherited from all parents of class C not in the set G_P. Set G is guaranteed to have at most one division for each selector.

3.2.4 Algorithms DSI and DCI (Determine Selector/Class Index)

Algorithm DSI is called to obtain a selector index, given a class selector pair. If the selector already has an index, the algorithm must determine whether a selector index conflict exists, and if so, compute a new index, store the index, allocate space in the table to handle the new index, and move all divisions for the selector from their old positions in the table to their new positions.

```
Algorithm DSI(inout $\sigma$ : Selector, in C : Class, inout T : DivisionTable)
1     $L_{old}$ := index($\sigma$)
2     if $L_{old}$ is unassigned or a selector index conflict exists
3           $L_{new}$ := indexFreeFor( classesUsing($\sigma$) $\cup$ dependentClasses(C,$\sigma$) )
4           index( $\sigma$ ) := $L_{n}ew$
5           if $L_{old} \neq$ unassigned then
6                 for $C_i \in classesUsing(\sigma)$ do
7                       T[$L_{new}, C_i$] := T[$L_{old}, C_i$]
8                       T[$L_{old}, C_i$] := $\Omega$
9                 endfor
10          endif
11    extend selector dimension of table to handle $L_{new}$
12    index($\sigma$) := $L_{new}$
13    endif
end DSI
```

In line 3, the function *indexFreeFor* is a dispatch-technique dependent algorithm that obtains an index that is not currently being used for any class that is currently using σ, as well as those classes that are dependent classes of $< C, \sigma >$. The algorithm is responsible for allocating any new space in the table necessary for the new index.

In line 5, if the old index is unassigned there are no divisions to move, since no divisions for σ currently exist in the table. Otherwise, the divisions for σ have changed location, and must be moved. The old locations are initialized with empty divisions.

Algorithm DCI is trivial, and is not presented.

4 The DT Framework

The DT Framework provides a collection of abstract classes that define the data and functionality necessary to modify dispatch information incrementally during environment modification. Recall that, from the perspective of the DT Framework, *environment modification* occurs when selectors or class hierarchy links are added or removed.

The DT Framework consists of a variety of special purposes classes [3]. Figure 5 shows the class hierarchies. We describe the data and functionality that each class hierarchy needs from the perspective of inheritance management and dispatch table modification. Clients of the framework can specify additional data and functionality by subclassing some or all of the classes provided by the framework.

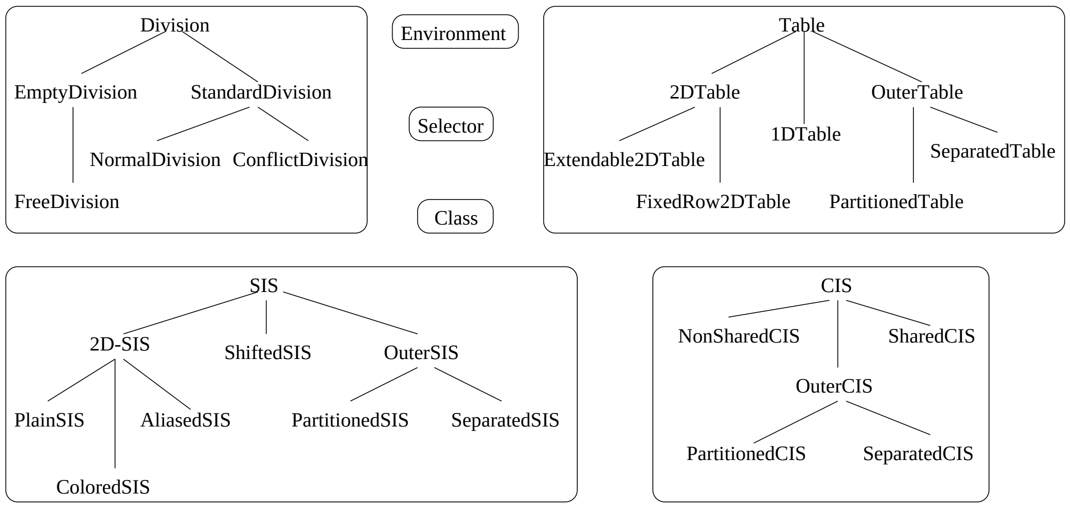


Figure 5: The DT Framework Class Hierarchy

4.1 The DT Classes

The Environment, Class and Selector classes are not subclassed within the DT Framework itself, but the Division, Table, SIS and CIS classes are subclassed. In the following subsections, we discuss the purpose of each class.

4.1.1 Environment, Class and Selector

The DT Environment class acts as an interface between the DT Framework client and the framework itself. However, since the client can subclass the DT Framework, the interface is a white box, not a black one. Each client creates a unique instance of the DT Environment and as class and method declarations are parsed (or evaluated at run-time), the client informs the Environment instance of these environment modifications by invoking its interface operations. These interface operations are: AS (Add Selector), RS (Remove Selector), ACL (Add Class Links), and RCL (Remove Class Links). The environment also provides functionality to register selectors and classes with the environment, save division

[3]In this discussion, we present the conceptual names of the classes, rather than the exact class names used in the C++ implementation.

tables, convert division tables to dispatch tables, merge division tables together and perform actual dispath for a particular class/selector pair.

Within the DT Framework, instances of Selector need to maintain a name. They do not maintain indices, since such indices are table-specific. Instances of Class maintain a name, a set of native behaviors, a set of immediate superclasses (parent classes), and a set of immediate subclasses (child classes). They provide mechanisms to add to, remove from and iterate over the sets. Furthermore, they implement an efficient mechanism for determining whether another class is a subclass or not.

4.1.2 Divisions

The Division hierarchy is in some ways private to the DT Framework, and language implementors that use the DT Framework will usually not need to know anything about these classes. However, divisions are of critical importance in providing the DT Framework with its incremental efficiency and compile-time method determination. Each division represents a method to be executed for a particular class/selector pair, and the class is referred to as the *defining class* of the division. Divisions, and their role in division tables, was discussed in Section 3.1.

4.1.3 Tables

Each Table class provides a fundamental structure for storing divisions, and maps the indices associated with a class/selector pair to a particular entry in the table structure. Each of the concrete table classes in the DT Framework provides a different underlying table structure. The only functionality that subclasses need to provide is that which is dependent on the structure. This includes table access, table modification, and dynamic extension of the selector and class dimensions of the table.

The 2DTable class is an abstract superclass for tables with orthogonal class and selector dimensions. Rows represent the selector dimension, and columns represent the class dimension. The Extendable2DTable class can dynamically grow in both selector and class dimensions as additional elements are added to the dimensions. The FixedRow2DTable dynamically grows in the class dimension, but the size of the selector dimension is established at the time of table creation, and cannot grow larger.

The 1DTable class is concrete, and represents tables in which selectors and classes share the same dimension. Selector and class indices are added together to establish an entry within this one dimensional table.

The OuterTable class is an abstract superclass for tables which contain subtables. Most of the functionality of these classes involves requesting the same functionality from a particular subtable. For example, requesting the entry for a class/selector pair involves determining (based on selector index) which subtable is needed, and requesting table access from that subtable. Individual selectors exist in at most one subtable, but the same class can exist in multiple subtables. For this reason, class indices for these tables are dependent on selector indices (because the subtable is determined by selector index). For efficiency, selector indices are *encoded* so as to maintain both the subtable to which they belong, as well as the actual index within that subtable. The PartitionedTable class has a dynamic number of FixedRow2DTable instances as subtables. A new FixedRow2DTable instance is added when a selector cannot fit in any existing subtable. The SeparatedTable class has two subtables, one for *standard selectors* and one for *conflict selectors*. A standard selector is one with only one root division (a new selector is also standard), and a conflict selector is one with more than one root division. Each of these subtables can be an instance of either Extendable2DTable or PartitionedTable. Since PartitionedTables are also outer tables, such implementations express tables as subtables containing subsubtables.

4.1.4 Selector Index Strategy – SIS

Each table has associated with it a selector index strategy, which is represented as an instance of some subclass of SIS. The OuterTable and 1DTable classes have one particular selector index strategy that they must use, but the 2DTable classes can choose from any of the 2D-SIS subclasses.

Each subclass of SIS implements Algorithm DSI (Determine Selector Index), which provides a mechanism for determining the index to associate with a selector, given a class/selector pair. Each SIS class maintains the current index for each selector, and is responsible for detecting selector index conflicts. When such conflicts are detected, a new index must be determined that does not conflict with existing indices. Algorithm DSI is responsible for detecting conflicts, determining a new index, storing the index, ensuring that space exists in the table for the new index, moving divisions from the old table locations to new table locations, and returning the selector index to the caller.

The abstract 2D-SIS class repesents selector index strategies for use with 2D-Tables. These strategies are interchangable, so any 2D-Table subclass can use any concrete subclass of 2D-SIS in order to provide selector index determination. The PlainSIS class is a naive strategy that assigns a unique index to each selector. The ColoredSIS and AliasedSIS classes allow two selectors to share the same index as long as no class in the environment recognizes both selectors. They differ in how they determine which selectors can share indices. AliasedSIS is only applicable to languages with single inheritance.

The ShiftedSIS class provides selector index determination for tables in which selectors and classes share the same dimension. This strategy implements a variety of auxillary functions which maintain doubly-linked freelists of unused entries in the one-dimensional table. These freelists are used to efficiently determine a new selector index. The selector index is interpreted as a shift offset within the table, to which class indices are added in order to obtain a table entry for a class/selector pair.

The PartitionedSIS class implements selector index determination for PartitionedTable instances. When selector index conflicts are detected, a new index is obtained by asking a subtable to determine an index. Since FixedRow2D subtables of PartitionedTable instances are not guaranteed to be able to assign an index, all subtables are asked for an index until a subtable is found that can assign an index. If no subtable can assign an index, a new subtable is dynamically created.

The SeparatedSIS class implements selector index determination for SeparatedTable instances. A new index needs to be assigned when a selector index conflict is detected or when a selector changes status from standard to conflicting, or vice-versa. Such index determination involves asking either the standard or conflict subtable to find a selector index.

4.1.5 Class Index Strategy – CIS

Each table has associated with it a class index strategy, which is represented as an instance of some subclass of CIS. The OuterTable and 1DTable classes have one particular class index strategy that they must use, but the 2DTable classes can choose from either of the 2D-CIS subclasses.

Each subclass of CIS implements Algorithm DCI (Determine Class Index), which provides a mechanism for determining the index to associate with a class, given a class/selector pair. Each CIS class maintains the current index for each class, and is responsible for detecting class index conflicts. When such conflicts are detected, a new index must be determined that does not conflict with existing indices. Algorithm DCI is responsible for detecting conflicts, determining a new index, storing the index, ensuring that space exists in the table for the new index, moving divisions from old table locations to new table locations, and returning the class index to the caller.

The NonSharedCIS class implements the standard class index strategy, in which each class is assigned a unique index as it is added to the table. The SharedCIS class allows two or more classes to share the same index if all classes sharing the index have exactly the same division for every selector in the table.

The PartitionedCIS and SeparatedCIS classes implement class index determination for PartitionedTable and SeparatedTable respectively. In both cases, this involves establishing a subtable based on the selector index and asking that subtable to find a class index.

5 Incremental Table-based Method Dispatch

All of the table-based techniques can be implemented using the DT Framework. However, due to the non-incremental nature of the virtual function table technique (VTBL), an incremental implementation of VTBL would be quite inefficient, so the current implementation of the framework does not support VTBL dispatch. All other techniques are provided, and the exact dispatch mechanism is controlled by parameters passed to the DT Environment constructor. The parameters indicate which table(s) to use, and specify the selector and class index strategies to be associated with each of these tables.

1. *STI*: uses Extendable2DTable, PlainSIS, and NonSharedCIS.

2. *SC*: uses Extendable2DTable, ColoredSIS, and NonSharedCIS.

3. *RD*: uses 1DTable, ShiftedSIS and NonSharedCIS.

4. *CT*: uses a SeparatedTable with two PartitionedTable subtables, each with FixedRow2DTable subsubtables. The selector index strategy for all subsubtables of the standard subtable is AliasedSIS, and the strategy for all subsubtables of the conflict subtable is PlainSIS. All subsubtables use SharedCIS.

5. *SCCT*: uses a SeparatedTable with two PartitionedTable subtables, each with FixedRow2DTable subsubtables. All subsubtables use ColoredSIS and SharedCIS.

The framework provides an elegant mechanism by which SC and CT dispatch can be merged into a new hybrid technique, SCCT. Using both SC and CT together works well to extend CT dispatch to languages with multiple inheritance. The *selector aliasing* performed by CT can remove all internal empty entries from each column of the table, without concern for other columns, as long as the table contains only standard selectors and uses only single inheritance. However, using selector coloring will often provide exactly the same compression (and will have a very small amount of internal empty space even in the worst cases), and is not restricted to single inheritance. Furthermore, CT dispatch cannot use selector aliasing to compress the conflict table, so all conflict selectors are effectively placed in an STI style table (although partitioning and class sharing can still be applied to this table to compress it). Using the SC technique on the conflict table provides the potential for more compression (note however, that coloring may reduce the amount of class sharing possible). The SCCT technique has the same dispatch performance as CT, slightly better space performance than CT, and generalizes CT to languages with multiple inheritance.

In addition to providing each of the above dispatch techniques, the framework can be used to analyze the various compression strategies introduced by CT dispatch in isolation from the others. For example, a dispatch table consisting of a PartitionedTable, whose FixedRow2DTable subtables each use PlainSIS and SharedCIS indexing strategies, allows us to determine how much table compression is obtained by class sharing alone. Many variations based on SeparatedTable and PartitionedTable, their subtables, and the associated index strategies, are possible.

6 Efficiency issues within Compilers and Run-time Systems

Both compilers and run-time systems benefit equally from the dispatch technique independence provided by the DT Framework. In addition, the framework provides each of them with powerful functionality.

6.1 Compilers

The DT Framework provides compilers with the following advantages: 1) maintenance of inheritance conflicts, 2) compile-time method determination, and 3) the ability to perform separate compilation.

In languages with multiple inheritance, it is possible for inheritance conflicts to occur, when a class with no native definition for a selector inherits two distinct methods for the selector from two or more superclasses. For the purposes of both efficiency and software verification, compile-time detection of such conflicts is highly desirable.

The most substantial benefit that the DT Framework provides to compilers is the recording of information needed to efficiently determine whether a particular class/selector pair is uniquely determined at compile-time. In such cases, the compiler can avoid run-time method dispatch entirely, and generate an immediate function call or even inline the code.

Another powerful capability provided to compilers by the DT Framework is separate compilation. Each library or collection of related classes can be compiled, and a full division table stored with the associated object code. At link-time, a separate DT Environment for each library or module can be created from the stored division tables. The linker can then pick one such environment (usually the largest) and ask that environment to merge each of the other environments into itself. This facility is critical in situations where a library is being used for which source code is not provided. Since certain dispatch table techniques require the full environment in order to maintain accurate tables (i.e. SC, RD and CT) library providers who do not want to share their source code need only provide the inheritance hierarchy and selector definition information needed by the DT Framework.

Finally, note that although it is necessary to use the extended dispatch table (i.e. a division table) to incrementally modify the inheritance information, in non-reflexive compiled languages it is not necessary to maintain the division table at run-time. Once linking is finished, the linker can ask the DT Environment to create a dispatch table from the division table, and this dispatch table can be stored in the executable for static use at run-time.

6.2 Run-time Systems

The DT Framework provides run-time systems with: 1) table-based dispatch in reflexive languages, 2) dynamic schema evolution, and 3) inheritance conflict detection.

The utility of the DT Framework is fully revealed when it is used by run-time systems. Because of the efficiency of incremental inheritance propogation and dispatch table modification, it can be used even in heavily reflexive languages like Smalltalk ([GR83]) and Tigukat ([OPS$^+$95]). However, this functionality is provided at the cost of additional space, because the entire division table must be maintained at run-time, rather than just a dispatch table containing addreses. Note also that without additional space utilization, division-table based dispatch is more expensive than normal table dispatch because of the indirection through the division stored at a division table entry in order to obtain an address. By doubling the table size, this can be avoided by having the division table store both a division pointer and an address. In dispatch techniques like RD and CT that are space-efficient, this doubling of size may be worth the improvements in dispatch performance.

Some mechanism to support dynamic schema evolution is necessary to provide languages with true reflexivity. The DT Framework allows arbitrary class hierarchy links to be added and removed no matter what the current state of the classes.

Finally, the framework allows inheritance conflicts to be detected at the time they are produced, rather than during dispatch. This allows reflexive languages to return error indicators immediately after a run-time environment modification. A common complaint with reflexivity is a lack of software verification; the DT Framework provides partial a solution to this.

7 Performance Results

In this section, we present some performance results obtained by applying the DT Framework to the class libraries described in Table 9. In the table, C is the total number of classes, S is the total number of selectors, M is the total number of legitimate class-selector combinations, m is the total number of defined methods, P is the average number of parents per class, and B is the size of the largest complete behavior, (c.f. [DH95]).

Library	C	S	M	m	P	B
Digitalk ST/V 2.0	534	4490	154616	6858	1	677
Parcplace1	774	5086	178230	8540	1	401
Geode	1318	6549	302709	14194	2.11	795
IBM Smalltalk 2.0	2320	14064	485779	26017	1	656
Parcplace2	1956	12583	578309	22968	1	514
Digitalk ST/V 3.0	1356	10062	613676	17104	1	1065
Self System 4.0	1801	10121	1038573	29414	1.02	966
Visual Age 2.0	3241	17404	1045949	37080	1	745

Table 9: Statistics for various object-oriented environments

In order to obtain the statistics presented in this section, a simple driver program was written which creates an instance of the DT Environment and parses an input file. Each line of the input file contains one of four directives (add/remove a selector for a class, or add/remove class hierarchy links). Thus, each line results in the invocation of one of the four DT Environment interface algorithms: AS, RS, ACL or RCL. Timings presented here are in milliseconds, and refer to the total user and system time taken to parse the entire input file and incrementally build a division table for the environment. The experiments were performed on a SparcStation-20/50 with 160Mb of RAM running SunOS4.1.4. The DT source code was compiled using g++ -O2. Some caveats on the timings should be noted. Relative performance results, in terms of execution speed, between the various dispatch techniques, are not representative of the fastest possible times. In general, none of the techniques have been optimized, and it is expected that a careful profiling will reveal many ways in which the overall framework, and the specific dispatch technique implementations, can be improved. On the other hand, fill-rate performance between techniques is optimal, but is discussed elsewhere ([VH96, DH95]) so is not readdressed here.

Not surprisingly, the order in which the environment is parsed can have a substantial effect on both execution performance and dispatch table fill-rate, given the incremental nature of the DT algorithms. In order to measure this effect, each of the library environments of Table 9 was ordered in multiple ways, and the DT algorithms were run on each input variation to establish timings and fillrates. From these experiments, it is possible to establish the optimal ordering for storing static libraries, as well as indicate how expensive random orderings are in reflexive languages. We have divided each input ordering using a *primary ordering* and a *secondary ordering*. The primary ordering determines how class definitions and selector definitions are intermixed. Native selectors can be defined immediately after each class definition, all selector definitions can occur after all class definitions, or all class definitions can occur after all selector definitions. Within each primary ordering, a secondary ordering establishes the order in which individual items (classes or selectors) appear. Classes can be ordered top-down, bottom-up or randomly. Selectors can occur by ordering the classes in various ways and and putting all native selectors for each class together, or can be grouped according to name (all selectors of the same name appear together). The DT Framework has been tested on the following input orderings:

1. *CSD*: classes are ordered top-down and all native selectors for each class occur immediately after the class definition
2. *CSU*: classes are ordered bottom-up and all native selectors for each class occur immediately after the class definition
3. *CSR*: classes are ordered randomly and all native selectors for each class occur immediately after the class definition
4. *CDSD*: all class definitions occur before any selector definition. Classes are defined by ordering them top-down. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for others classes in the ordering.
5. *CDSU*: like CDSD except selectors are defined by ordering classes bottom-up and putting all native behaviors for each class in this order together.

6. *CDSR*: like CDSD except selectors are defined by ordering classes randomly and putting all native behaviors for each class in this order together.

7. *CUSD*: all class definitions occur before any selector definition. Classes are defined by ordering them bottom-up. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for other classes in the ordering.

8. *CUSU*: like CUSD except selectors are defined by ordering classes bottom-up and putting all native behaviors for each class in this order together.

9. *CUSR*: like CUSD except selectors are defined by ordering classes randomly and putting all native behaviors for each class in this order together.

10. *CRSD*: all class definitions occur before any selector definition. Classes are defined by ordering them randomly. The order in which selectors appear is determined by ordering classes top-down and defining all native selectors for each class in this ordering together, before native selectors for other classes in the ordering.

11. *CRSU*: like CRSD except selectors are defined by ordering classes bottom-up and putting all native behaviors for each class in this order together.

12. *CRSR*: like CRSD except selectors are defined by ordering classes randomly and putting all native behaviors for each class in this order together.

13. *RDD*: all classes are defined before any selector, and classes are ordered top-down. All definitions for the same selector occur together, and selectors occur by sorting them in descending order based on the number of classes that recognize them (i.e., selectors recognized by more classes are defined before those recognized by fewer). Note that the RDD ordering is the closest to the optimal ordering identified by [DH95] for RD dispatch.

14. *RDU*: like RDD except that classes are ordered bottom-up (selectors appear in the same order they do in RDD).

15. *RND*: the totally random ordering – the order of class and selector definitions is completely random.

Due to the number of combinations possible, we do no present results for every combination of dispatch technique, library and input ordering in this paper. Instead, we have choosen representative examples. For the most part, we will focus on SC dispatch and the Parcplace1 library, whose graphs are, for the most part, representative of other techniques and libraries.

The results have been divided into two subsections. In the first, we determine which input ordering provides the best execution time and fill-rate performance. This is useful because all object-oriented languages, reflexive or not, provide code reuse via libraries. The DT algorithms can be used to create a division table for each library. This division table would be stored with the library and loaded as the initial division table when application code is to be compiled. Thus, application code would incrementally modify a precomputed division table. The time taken for the DT algorithms to create a division table for a library represents the amount by which compilation would slow down if the DT algorithms were used by the compiler. The second subsection presents results on the effects of random input orderings on execution time and fill-rate, including per-modification timings. These timings represent how long the execution of a run-time system is delayed each time a selector or class is added at run-time.

7.1 Static Input Orderings

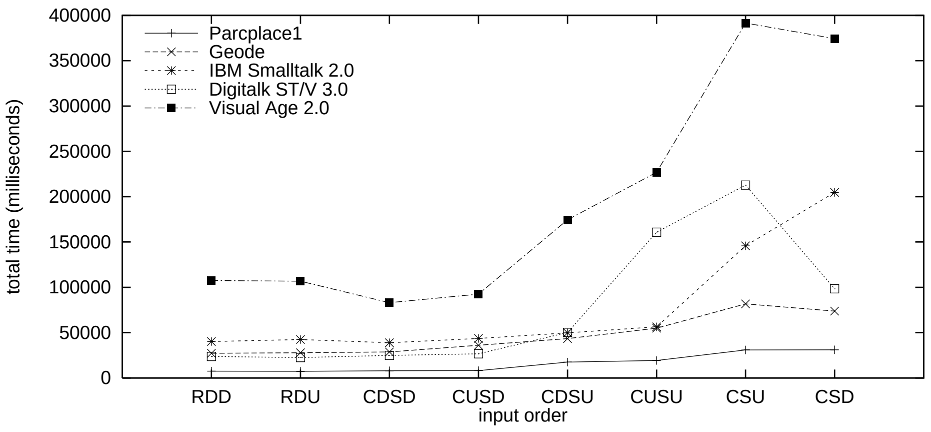


Figure 6: Input order vs. Execution time for SC dispatch

There are two ways in which input order affects execution time. First, certain orderings will require less inheritance propogation than others. For example, an input ordering in which selectors are defined based on top-down class order will require much more inheritance propogation than an ordering in which selectors are defined based on bottom-up class ordering (the former order must propogate divisions that are subsequently overridden). Second, certain orderings will require fewer calls to Algorithm DSI. Since Algorithm DSI is usually the most expensive algorithm in the DT Framework, avoiding it is desirable. Unnecessary calls to Algorithm DSI can be avoided by ordering the environment so that selectors appear based on top-down class order. In this way, the first call to DSI will find an index free for the largest number of dependent classes. In the opposite order, with selectors appearing based on bottom-up class order, indices are assigned based on only a small number of the classes that will eventually recognize the selector, requiring additional calls to DSI as selector definitions for classes higher in the hierarchy are obtained. Note that the two manners in which input order affect execution time compete with one another. One is minimized by selectors ordered by classes top-down, and the other by selectors ordered by classes bottom-up.

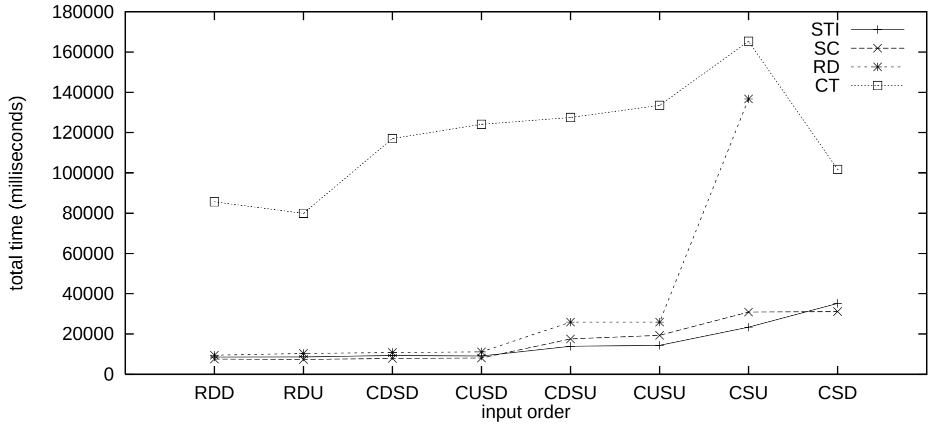


Figure 7: Input order vs. Execution time for Parcplace1

Figure 6 shows the time, in milliseconds, taken by the DT Framework to create a selector colored division table (SC), using each of the non-random input orderings. From the graph, we can make the following conclusions. RDD, RDU, CDSD and CUSD are roughly equal (which is better depends on the library being processed). All of these are slightly better than CDSU and CUSU, which are usually much better than CSD and CSU. These overall trends hold true across all techniques, although the degree by which timings are affected varies with technique. Figure 7 shows the effects of input order on execution time for each of STI, SC, RD and CT[4] on the Parcplace1 library. Results for SCCT are not shown because they are almost identical to CT.

Input ordering has a slightly different effect on fill-rate. Figure 8 shows fillrates for the non-random input orderings using SC dispatch, and Figure 9 shows fillrates for all four of the dispatch techniques when these input orderings are applied to the Parcplace1 library.

Input orders RDD and RDU provide the best fill-rates, followed by CDSD, CUSD and CSD (unlike for execution times, where CSD was worst). The bottom-up selector orderings (CDSU, CUSU and CSU) give the worst fill-rates. Notice that, from a fill-rate perspective, RD dispatch is most sensitive to input ordering, and STI dispatch is not affected at all. Remember that RDD/RDU represent the input ordering identified by [DH95] as optimal for fill-rate performance in RD dispatch.

From the previous graphs, we can conclude that the best possible ordering for both execution time and maximal fill-rate is RDD or RDU. Exactly which

[4]The results reported here are for a version of CT in which selector coloring is used instead of selector aliasing.

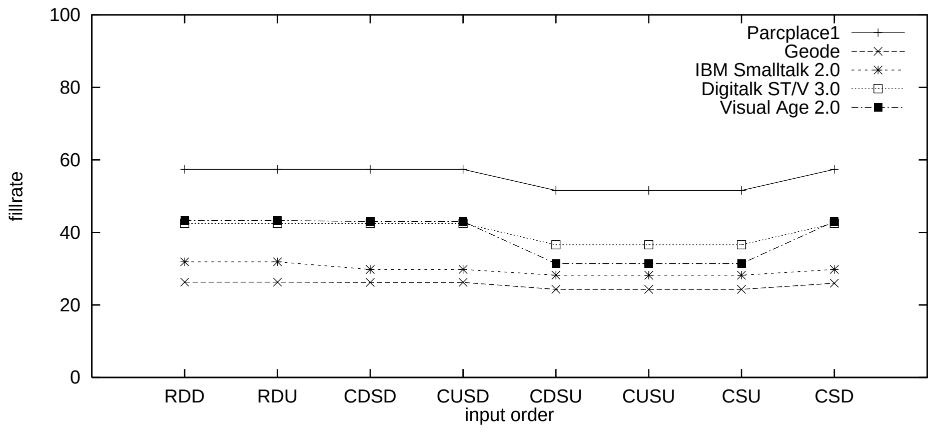


Figure 8: Input order vs. Fill-Rate for SC dispatch

one is better varies on the dispatch technique, library and input order, but, on average, RDD gives the best results.

7.2 Random Orderings

Knowing the optimal static ordering is useful in determining how library code should be stored to make recomputation of a library division table optimal. However, in reflexive languages, such fine control over input ordering is not possible. In order to determine how the DT Framework performs on random input, we generated 10 versions of each of the random orderings. The average execution time and fill-rate across these 10 input files gives a good measure of the performance of the algorithms on random data. Figures 10 and 11 show the execution time and fill-rate performance respectively for some of these random orderings. We have also included some non-random orderings for comparison. The totally random ordering, RND, is approximately 2.5 times slower than the optimal ordering, RDD, and twice as fast as the worst ordering, CSD.

In reflexive environments, the per-invocation cost of the incremental algorithms is also of interest. Figures 12 and 13 show the average time (in milliseconds) of a call to Algorithm AS and ACL respectively.

The average per-invocation cost of adding a selector in environments with about half a million class/selector pairs is approximately one millisecond. The average per-invocation cost of adding class hierarchy links is at most 80 milliseconds. Note that although order CSD is optimal for Algorithm AS, it is the absolute worst ordering for Algorithm ACL. In this ordering, no inheritance propogation occurs during Algorithm AS, and redundant inheritance propogation occurs during Algorithm ACL. As expected, the best overall ordering is RDD. During Algorithm AS, the truly random ordering, RND, is not much more expensive than RDD. However, during Algorithm ACL, the random ordering is

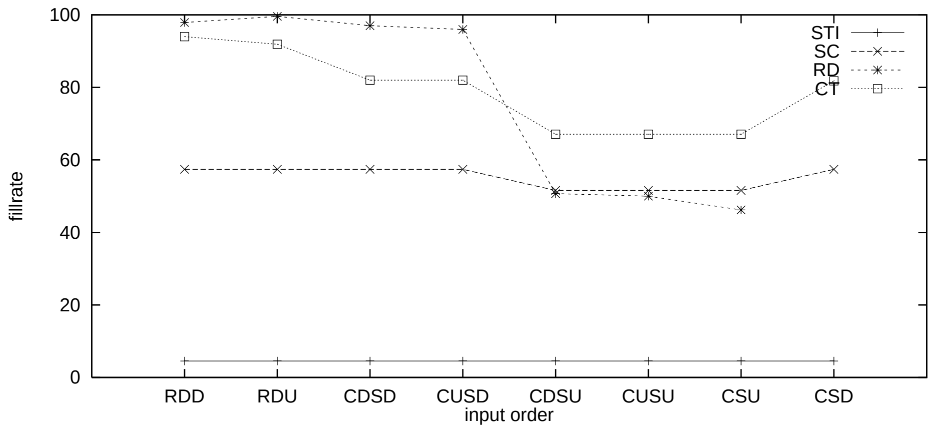


Figure 9: Input order vs. Fill-Rate for Parcplace1

much more expensive than order RDD, but is about 75% more efficient than order CSD.

8 Related and Future Work

8.1 Related Work

[DHV95] presents an analysis of the various dispatch techniques and indicates that in most cases, IC and PIC are more efficient than STI, SC and RD, especially on highly pipelined processors, because IC and PIC do not cause pipeline stalls that the table indirections of STI, SC and RD do. However, even if the primary dispatch technique is IC or PIC, it may still be useful to maintain a dispatch table for cases were a miss occurs, as a much faster alternative to using ML (method lookup) or LC (global cache) and ML together. Especially in reflexive languages with substantial multiple inheritance, ML is extremely inefficient, since each inheritance path must be searched (in order to detect inheritance conflicts).

[DGC95] discusses static class hierarchy analysis and its utility in optimizing object-oriented programs. They introduce an *applies-to* set representing the set of classes that share the same method for a particular selector. These sets are represented by our concept of dependent classes. Since each division implicitly maintains its set of dependent classes, the DT algorithms have access to such sets, and to the compile-time optimizations provided by them.

[AR92] presents an incremental approach to selector coloring. However, the algorithm proposed often performs redundant work by checking the validity of selector colors each time a new selector is added. The DT algorithms demonstrates how to perform selector color determination only when absolutely necessary (i.e. only when a selector color conflict occurs), and generalize the ap-

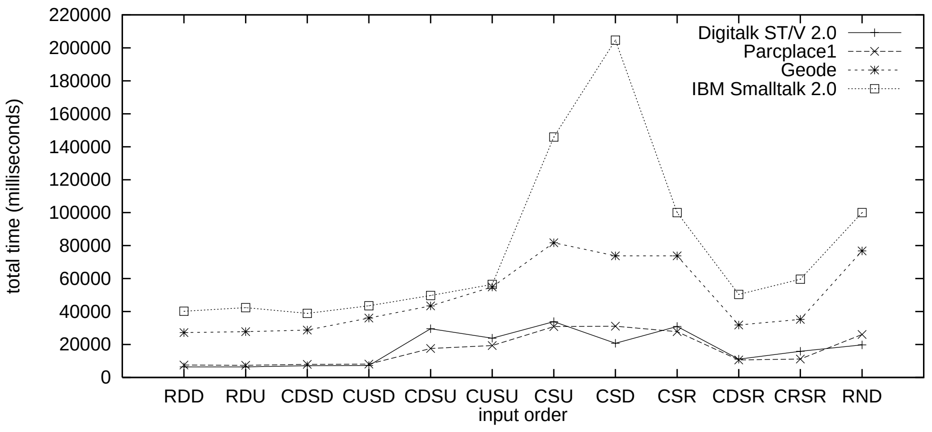


Figure 10: Random Input Order vs. Execution Time for Parcplace1

proach to a variety of table-based approaches. [DH95] presents selector-based row displacement (RD) and discusses how to obtain optimal compression results. [VH96] presents the compact selector indexed table (CT), expanding on previous work in [VH94].

Predicate classes, as implemented in Cecil ([Cha93]), allow a class to change its set of superclasses, at run-time. The DT Framework provides an efficient mechanism for implementing predicate classes using table-based dispatch.

8.2 Future Work

The DT Framework provides a general description of all work that needs to be performed to handle inheritance management and method dispatch in reflexive, dynamically typed, single-receiver languages with multiple inheritance. A variety of extensions are possible.

First, the framework as presented handles methods, but not internal state. A mechanism to incrementally modify object layout is a logical, and necessary, extension. Second, multi-method languages such as Tigukat [OPS+95] and Cecil [Cha92] have the ability to dispatch a method based not only on the dynamic type of a receiver, but also on the dynamic types of all secondary arguments to the behavior. Multi-methods extend the expressive power of a language, but efficient method dispatch and inheritance management is an even more difficult issue in such languages. Third, the framework currently assumes that inheriting the interface of parent classes implies that the implementation assocated with the interface is inherited also. A more general mechanism for inheritance management that separates these concepts is desirable.

Fourth, although the DT Framework provides a general mechanism for handling table-based method dispatch, it is really only one component of a much larger framework that handles all method dispatch techniques. The DT Frame-

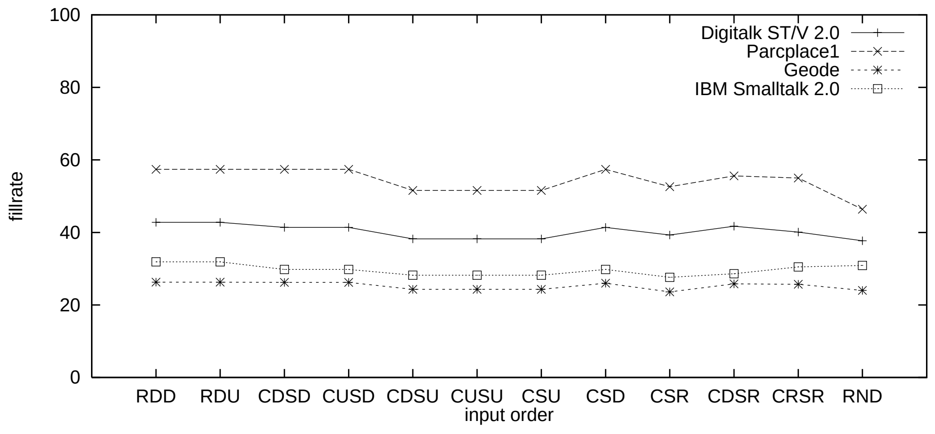


Figure 11: Random Input Order vs. Fill-Rate for Parcplace1

work can be extended so that framework clients call interface algorithms each time a call-site is encountered, similar to the manner in which the environment is currently called, when class and selector definitions are encountered.

Fifth, research into the impact on dispatch performace incurred by the dynamic nature of the DT Framework is needed. Note that in the framework, the fundamental structures like arrays and sets (used by many classes) are grown dynamically, with additional space being added as necessary. Unfortunately, this capability to dynamically extend selector and class dimensions within tables necessitates additional indirections during table access, making actual dispatch less efficient than non-dynamic implementations of the same table-based techniques. As observed in [DHV95], the table-based dispatch techniques are more inefficient than dynamic techniques like IC and PIC, since table-based dispatch techniques cause pipe-line stalls. Since one indirection stalls a pipe as effectively as two or three indirections, it is expected that the extra indirections incurred by the dynamic nature of the division tables will not substantially reduce dispatch times.

Sixth, the DT Framework allows various compression techniques, like selector aliasing, selector coloring, and class sharing, to be analyzed both in isolation, and in interaction with one another. More research about how these techniques interact, and about how SCCT dispatch can be optimized, is necessary.

9 Conclusion

We have presented a framework that is usable by both compilers and run-time systems to provide table-based method dispatch, inheritance conflict detection, and compile-time method determination. The framework relies on a collection of technique independent algorithms for environment modification, which call technique-dependent algorithms to perform fundamental operations like table

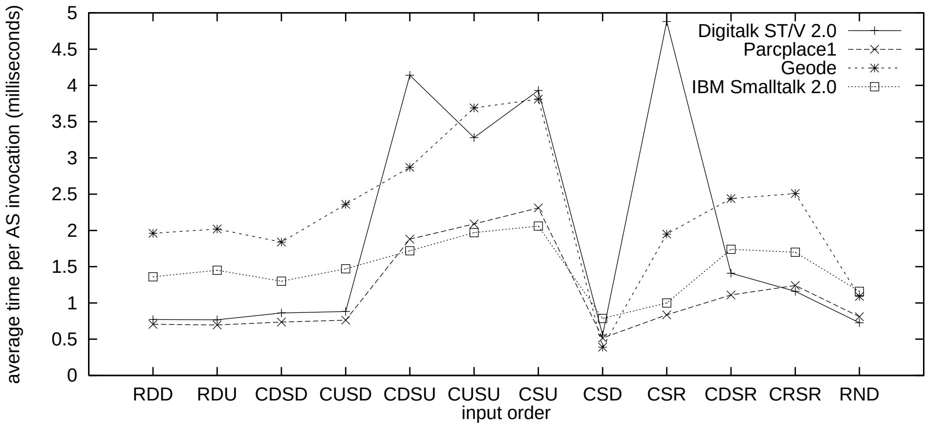


Figure 12: Cost of Algorithm AS Invocation

access and index determination. The framework unifies all table-based method dispatch techniques into a cohesive whole, allowing a language implementor to change between techniques by changing the manner in which the DT Environment is instantiated. Incremental versions of all table-based techniques except VTBL have been implemented, all of which have low milli-second per-modification execution times.

The framework provides a variety of new capabilities. The various table-based dispatch techniques have differing dispatch execution times and memory requirements. Since the framework allows any table-based dispatch technique to be used, a particular application can be optimized for either space or dispatch performance. Furthermore, the DT Framework allows table-based dispatch techniques to be used in reflexive languages. In the past, reflexive languages necessitated the use of a non-table-based techique. One reason that C++ uses virtual function tables is that they allow for separate compilation, unlike other table-based dispatch techniques. The DT Framework now allows all table-based dispatch techniques to work with separate compilation. Finally, the framework introduces a new level of software verification in reflexive languages by allowing inheritance conflicts to be detected immediately when they occur, rather than during dispatch.

The framework has been used to merge SC and CT method dispatch into a hybrid dispatch technique with the advantages of both. The CT dispatch technique is limited by its restriction to single-inheritance. By replacing selector aliasing by selector coloring, we obtain a dispatch technique that works with multiple inheritance and that benefits from the class sharing made possible by CT partitioning. Furthermore, SCCT dispatch provides slightly better compression because the conflict table can be colored, unlike in CT dispatch, where it remains uncompressed.

As an indication of the efficiency of our algorithms and implementation,

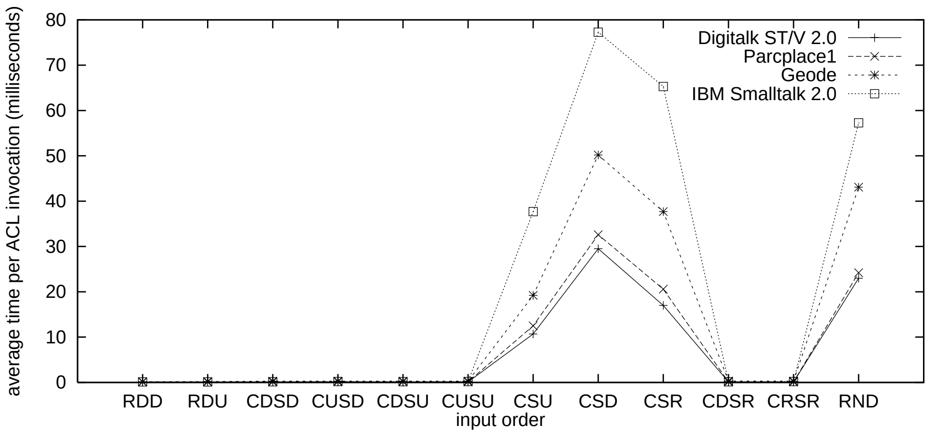


Figure 13: Cost of Algorithm ACL Invocation

[AR92] reports a time of 12 minutes to color the Smalltalk-80 Version 2.5 hierarchy when the 106 selectors native to Object are ignored, using a Sun 3/80. Our implementation of the DT algorithms color the entire library in 113 seconds on a Sun 3/80, while performing inheritance detection and maintaining compile-time optimization information. The DT Framework currently consists of 36 classes, 208 selectors, 494 methods, and 1081 meaningful class/selector pairs. When the DT Framework is applied to a completely random ordering of itself, a SCCT-based dispatch table is generated in 0.436 seconds. Since compiling the framework requires 390 seconds, even the slowest dispatch technique and input ordering produce a dispatch table in a negligible amount of time, relative to overall compilation time.

References

[AR92] P. Andre and J.C. Royer. Optimizing method search with lookup caches and incremental coloring. In *OOPSLA'92 Conference Proceedings*, 1992.

[Cha92] Craig Chambers. Object-oriented multi-methods in cecil. In *ECOOP'92 Conference Proceedings*, 1992.

[Cha93] Craig Chambers. Predicate classes. In *ECOOP'93 Conference Proceedings*, 1993.

[Cox87] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, 1987.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimzation of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 Conference Proceedings*, 1995.

[DH95] K. Driesen and U. Holzle. Minimizing row displacement dispatch tables. In *OOPSLA'95 Conference Proceedings*, 1995.

[DHV95] K. Driesen, U. Holzle, and J. Vitek. Message dispatch on pipelined processors. In *ECOOP'95 Conference Proceedings*, 1995.

[DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA'89 Conference Proceedings*, 1989.

[Dri93a] Karel Driesen. Method lookup strategies in dynamically typed object-oriented programming languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[Dri93b] Karel Driesen. Selector table indexing and sparse arrays. In *OOPSLA'93 Conference Proceedings*, 1993.

[DS94] L. Peter Deutsch and Alan Schiffman. Efficient implementation of the smalltalk-80 system. In *Principles of Programming Languages*, Salt Lake City, UT, 1994.

[ECO95] *ECOOP'95 Conference Proceedings*, 1995.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GR83] A. Goldberge and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[HCU91] Urs Holzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings*, 1991.

[HS96] Wade Holst and Duane Szafron. Inheritance management and method dispatch in reflexive object-oriented languages. Technical Report TR-96-27, University of Alberta, Edmonton, Canada, 1996.

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[OPS+95] M.T. Ozsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, , and A. Munoz. Tigukat: A uniform behavioral objectbase management system. In *The VLDB Journal*, pages 100–147, 1995.

[VH94] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method lookup for dynamically typed languages. In *ECOOP'94 Conference Proceedings*, 1994.

[VH96] Jan Vitek and R. Nigel Horspool. Compact dispatch tables for dynamically typed programming languages. In *Proceedings of the Intl. Conference on Compiler Construction*, 1996.

A Assertions

In Section 3.2.2, a variety of division and class concepts were introduced and used to obtain efficient tests for identifying which of three scenarios a particular invocation of Algorithm IMA or IMR should execute. Table 6 is a truth table showing all possible combinations of truth values for four fundamental tests. The following assertions allow us to eliminate most combinations of the tests as impossibilities. In the assertions, we use σ to represent $D_N.\sigma$.

1. C_T *is never nil*: From the definition of target class, C_T.

2. C_B *is never nil*: From the definition of base class, C_B.

3. $D.C = nil \Rightarrow D = \Omega$: The only division whose defining class is nil is the empty division, Ω. This is the definition of the representation of the empty division.

4. *In IMA,* $D_N \neq \Omega$: During method addition, such an empty division will never be propogated (Algorithm AS always creates a new division, and Algorithm ACL only propogates non-empty divisions). This implies that in Algorithm IMA, $D_N.C \neq$ nil and $D_N.\sigma \neq$ nil, from Assertion 3.

5. $C_T \leq C_B \leq C_N$: follows from the definition of these classes. $C_B \leq C_N$ is obviously only true when $C_N \neq nil$

6. *In IMR, C_I is never nil*: remember that C_I refers to the class from which C_T inherits σ, before σ is added/removed from C_B. During method removal, if the definition of σ in C_B is not visible to C_T it is because some class between C_B and C_T has redefined σ. In either case, C_T inherits σ from some real class and thus C_I cannot be nil.

7. *If $C_I \neq nil, C_T \leq C_I$*: It is not possible to inherit a method from a subclass, so since C_I is defined as the class from which C_T inherits σ before D_N is inserted, $C_T \leq C_I$, if such an inheriting class, C_I, exists.

8. $C_N = C_I \Rightarrow C_I \neq C_T$: Suppose not, so it is possible that $C_N = C_I = C_T$. However, in Algorithms ACL, RCL and RS, D_N is always associated with a class strictly above C_B in the inheritance hierarchy. Thus, our assumption is only possible from Algorithm AS. In this situation, Algorithm AS does not need to do any inheritance propogation whatsoever, since $D_C.C = D_N.C$ and $D_C.\sigma = D_N.\sigma$. Thus, this assertion is true because it is enforced to be true by our algorithms.

9. *$C_N \neq nil$ and $C_I \neq nil$ and C_N NLE $C_I \Rightarrow \pi$ is true*: First, note that C_N NLE $C_I \Rightarrow C_I < C_N$ or C_I and C_N are not orderable.

   (a) *Suppose C_I and C_N are not orderable*: By the definition of C_I, σ is visible in C_T from C_I before adding D_N. Since C_N NLE C_I, the new division does not block the visiblity of σ in C_T from C_I, so after the method addition, σ is visible in C_T from C_I. Similarily, after method addition, σ is visible in C_T from C_N because C_I NLE C_N. Thus, π is true.

   (b) *Suppose $C_I < C_N$*: Since $C_T \leq C_I$ (from 7), at least one path from C_N to C_T has C_I along it. Suppose all paths from C_N to C_T have C_I along them. Then C_T would never have been reached by the algorithm, because, on a previous invocation, the algorithm would have previously encountered the situation in which $C_T = C_I$, and recursion would have stopped. Since C_T has been reached, our supposition is incorrect, and there exists a path from C_N to C_T that does not pass through C_I, so σ is visible in C_T from C_N. Since $C_I < C_N$, there is a path from C_I to C_T that does not pass through C_N, so σ is visible in C_T from C_I. Thus, π is true.

10. *$C_I = nil \Rightarrow \pi$ is false*: $C_I = nil \Rightarrow sigma$ is not visible in C_T from C_I. Condition πrequires that σ be visible in C_T from both C_I and C_D.

11. *$C_N = C_I \Rightarrow \pi$ is false* : by the definition of π.

12. *In IMR, $C_T \neq C_I \Rightarrow C_I = C_B$*: Suppose $C_T \leq C_B < C_I$. Observe that there must exist a native definition for σ in C_B in order to be able to remove σ from C_B. Thus, before adding D_N, C_T would inherit σ from $C_B \neq C_I$, which contradicts the definition of C_I. Therefore, C_B NL C_I. Suppose $C_T < C_I < C_B$. Algorithm IMR is initially invoked on child

classes of C_B, and would stop recursion when it encountered a subclass with a native definition (i.e. when it encountered C_I). But this implies that C_T would never be reached (since $C_T < C_I$) unless there exists some other path from C_B to C_T that does not pass through C_I. However, if this were the case, σ would be visible in C_T from both C_B and C_I, implying that a conflict exists, in which case an implicit native definition representing a conflict would exist in C_T. This would mean that $C_T = C_I$, contradicting our initial assumption. Therefore, $C_I \; NL \; C_B$. Similarily, if C_I and C_B were unrelated in the inheritance hierarchy, σ would be visible in C_T from both C_I and C_B, and we have already shown that this is not possible, since $C_T \neq C_I$. The only remaining possibility is that $C_I = C_B$.

13. *In IMR, $C_T \neq C_I \Rightarrow \pi$ is false*: $C_T \neq C_I$ implies that, before D_N is added, there was no inheritance conflict (remember that an inheritance conflict results in an implicit native definition). D_N is either Ω(which can never cause an inheritance conflict) or a division defined in some superclass of $C_I = C_B$ (see Assertion 12). In the latter case, since the definition in C_I is being removed, after adding D_N, σ is not visible in C_T from C_I, so π is false. This implication says that it is not possible to create an inheritance conflict during method removal (i.e. during an invocation of IMR).

B Utility Algorithms

B.1 Algorithm RIC (Record Inheritance Conflict)

Algorithm RIC abstracts all the code necessary to record an inheritance conflict between two divisions.

```
Algorithm RIC(in $\sigma$ : Selector, in C : Class, in G : Set of Divisions) : Division
    G := G - $\Omega$

    if $normG > 1$ then
        if $\exists D \in Gst$ isConflict(D) and D.C = C then Note 1
            foreach $D_i \in G - \{D\}$ do
                addDivisionChild($D$,$D_i$)
            endfor
        else
            D := newConflictDivision($C$,$\sigma$) Note 2
            foreach $D_i \in G$ do
                addDivisionChild($D_i$, $D$)
            endfor
        endif

    return D
end RIC
```

B.1.1 Notes for Algorithm RIC

1. Division D already represents a conflict division for class C, so all other divisions in G are new parent divisions adding to an existing conflict. We make the appropriate division links. Only one such conflict division can possibly exist in G at any given time.

2. Algorithm *newConflictDivision* creates a new conflict division for class C and selector σ. It is trivial, and not presented here.

B.2 Algorithm ID (Inherited Division)

Algorithm ID (Inherited Division) obtains the division that would be inherited in class C for selector σ if a native definition did not exist and class C only had the classes in G_P as parents.

```
Algorithm ID(inout $\sigma$ : Selector, in C : Class, in $romanG_P$ : Set of Classes, in G : Set of Division, inout T : DivisionTable)
    foreach $C_i \in G_P$ do Note 1
        D := divisionFor($\sigma$,$C_i$)
        if not isEmpty(D) then
            add D to G
        endif
    endfor

    if $normG = 0$ then Note 2
        $D_N := \Omega$
    elsif $normG = 1$ then Note 3
        $D_N$ := the single element of G
    else Note 4
        $D_N$ := RIC($\sigma, C, G$)
    endif

    return $D_N$
end ID
```

B.3 Notes for Algorithm ID

1. Loop over all classes in specified parent set and obtain the non-empty divisions associated with them for σ. The resulting set, G, represents all methods visible in class C from parents in G_P. The procedure

2. *divisionFor(σ,C)* returns the division representing the address to be executed for selector σ and class C. In STI dispatch, this is identical to T[σ,C], but in SC and RD dispatch, the division obtained via T[σ,C] may not even represent σ (due to the table compression performed by these techniques. Thus, if $T[\sigma, C].\sigma \neq \sigma$, the procedure returns Ωinstead. The procedure is trivial, and is not presented.

3. If there are no parent divisions, removing the current selector means that the empty division should be stored in dependent classes of C.

4. If there is exactly one parent division, this parent division should be propogated to dependent classes of C.

5. If there are more than one parent divisions, an inheritance conflict has occured. Algorithm RIC is called to record this inheritance conflict, and the resulting conflict division is placed in the dependent classes of C.

B.4 Algorithm ICB (Inherited Class Behavior

Given a class, C, and a set of classes, G, Algorithm ICB returns the set of divisions that would be inherited from classes in G if each of these classes was a parent of class C. Since G can be a subset of the complete set of parents for class C, the division set returned will not, in general, constitute all inherited behavior. If a particular selector has both a native definition and a definition in a superclass, it is not included in the returned set (because it is not inherited in class C). However, in determining whether, for a given selector, a conflict exists, the algorithm considers the divisions for class C and all classes in G. If more than one division represents the same selector, a conflict for that selector is made and added to the set to be returned.

```
Algorithm ICB(in C : Class, in G : Set of Classes, in T : DivisionTable) : Set of Divisions
    H := Note 1
    foreach selector $\sigma$ do
        $D_C$ := divisionFor($\sigma$,$C$) Note 2
        if $D_C.C \neq$ C then
            D := ID($\sigma, C, G, \{D_C\}, T$) Note 3
            add $< \sigma, D >$ to H
        endif
    endfor

    return H
end ICB
```

B.5 Notes for Algorithm ICB

1. Set H will contain two-tuples as elements, where each tuple contains a selector and a division. The selector is redundant when the division is non-empty, but necessary when empty divisions need to be propogated (i.e. Algorithm IMR). The set is guaranteed to have only one tuple per selector.

2. The procedure *divisionFor(σ,C)* returns the division representing the address to be executed for selector σ and class C. In STI dispatch, this is identical to T[σ, C], but in SC and RD dispatch, the division obtained via T[σ, C] may not even represent σ (due to the table compression performed by these techniques. Thus, if T[σ, C].$\sigma \neq \sigma$, the procedure returns Ωinstead. The procedure is trivial, and is not presented.

3. Algorithm ID returns the division that would be inherited in class C for selector σ if no native definition existed in C and C only had the parents in G. It is presented in this appendix

C Compile-time Method Determination

This appendix summarizes how the DT algorithms can be used to determine when a method is uniquely identified at compile-time. Each class/selector pair is characterized in terms of its relation to other class/selector pairs in the environment. To this end, we define six mutually exclusive partition types that are useful for various purposes. Each class/selector pair $< C, \sigma >$ has one partion type.

1. *undefined*: σ has not been defined any class in the application. In Figure 1, $< F, \gamma >$ is undefined since γ is not defined in any of the application classes F..Q

2. *unrelated*: σ has been defined in at least one class in the application, but has not been defined in any class in the connected inheritance graph containing C. In Figure 1, $< F, \lambda >$ is unrelated since λ is not defined in any of the application classes F..I, but is defined in class M.

3. *sub-defined*: σ has been defined in at least one subclass of C, but has not been defined in C or any of its superclasses. In Figure 1, $< F, \beta >$ is sub-defined since β is defined in class G, but not in F.

4. *defined-determined*: σ is uniquely visible in C, but is not explicitly defined in any subclass of C. In Figure 1, $< K, \beta >$ is defineddetermined since β is defined in superclass J of K, but not in any subclass of K.

5. *defined-undetermined*: σ is uniquely visible in C and is defined in a subclass of C. In Figure 1, $< N, \lambda >$ is definedundetermined since λ is defined in superclass M and in subclass P of N.

6. *conflicting*: σ is multiply visible in C and C does not explicitly define σ. In Figure 1, $< M, \alpha >$ is conflicting since α is defined in both K and L.

At every call-site, the compiler knows the selector and the static type (class) of the receiver object. By asking the DT Environment for the partition type of this class/selector pair, the compiler can establish whether a unique method exists for the call-site. In particular, if the partition type is *defined-determined*, *undefined*, *conflicting*, or *unrelated*, a unique method exists.