EFFICIENT SUPPORT FOR OBJECT MUTATION AND TRANSPARENT FORWARDING

By Dave Thomas, Wilf Lalonde and John Duimovich

SCS-TR-128

November 1987

This research was supported in part by DREO.

School of Computer Science, Carleton University
Ottawa, Canada K1S 5B6

Efficient Support for Object Mutation and Transparent Forwarding

Extended Abstract

Dave Thomas, Wilf Lalonde and John Duimovich
School of Computer Science
Carleton University

Abstract

Distributed object oriented applications such as CAD/CIM require efficient support for object mutation and message forwarding. Many research prototypes have been implemented in Smalltalk using **doesNotUnderstand** to implement forwarders and **become:** to perform object mutation. The recent development of large object space smalltalks has pointed out the cost of the Smalltalk object mutation facility become:. Performance concerns have caused some implementers to recommend removing become: from the language, arguing that the construct is not really needed. In this paper we argue the need for a **become:** construct to support object mutation and describe simple techniques for efficiently implementing both object mutation and transparent forwarding.

Introduction

Distributed object oriented applications such as CAD/CIM require efficient support for object mutation and message forwarding. Object oriented systems provide a natural way to describe many manufacturing applications. These systems are distributed and must support migrating objects which model the underlying physical manufacturing process. They require alternative representations of both transient and persistent objects. Languages like Smalltalk, although originally designed for personal workstations are promising vehicles for implementing such systems. Smalltalk was not designed to support distributed objects with multiple representations, but it provides the basic facilities for prototyping such systems.

Current prototypes [Bennett87, McCullough87] have been implemented in Smalltalk using **doesNotUnderstand** to implement forwarders and **become:** to perform object mutation. Unfortunately both of these operations are relatively expensive in current Smalltalk implementations. In particular, the recent development of large object space smalltalks has pointed out the cost of the Smalltalk object mutation facility become:. Some researchers have advocated removing become: from the language because of its impact on performance. They also argue that it is an unnecessary construct.[Ungar86] In this paper we argue the need for a become: construct to support object mutation. We describe simple techniques which we have used[LTP86] to provide both object mutation and fast forwarding.

Why become: is Useful and Necessary

One of the major advantages of object oriented systems is their ability to encapsulate the representation of complex objects. In CIM for example we want to be able to have a compact(page) disk representation of a circuit board which can be quickly read from disk and transferred over a network. We also want a memory based representation which can be quickly and easily traversed by the board tester. These two exemplars[LTP86] are really instances of the same object undergoing a metamorphosis as they move through the factory. It is imperative that such change of representation occur without impacting existing code which references the object. Note that in this example the board goes through the following changes of representation (a) from a memory based forwarding pointer to a disk page, (b) from a disk page to a pointer connected structure, (c) from a linked structure to a disk page, and (d) from a disk page to a forwarding pointer.

Metamorphosis requires that we have an efficient and unobtrusive facility for object mutation. Smalltalk has such a facility called **become:**. It takes and object and mutates it to another object. The operation is atomic, such that in a single become: all references to the new representation are correctly updated. It has been used for many years to mutate fixed size data structures such as arrays into larger arrays. This facility allows programs to adjust dynamically to their changing environment. Without it programmers must either allocate sufficient array storage, or they must tolerate indirection overhead to access the elements of the array[WirfsBrock86]. The former technique wastes valuable main memory storage due to over allocation or fails due to under allocation. The latter impacts the performance of array references but is certainly an improvement.

One-Way Versus Two-Way Become:

At the implementation level, all variables in Smalltalk are pointers. This applies not only to local variables and parameters but also to instance variables in objects. An assignment such as A ← B which can be paraphrased as "A is bound to the same object that B is bound to" is implemented using pointer assignment. It is important to realize that the object referenced by B is not copied. This might best be described it as a **binding semantics** to differentiate it from **copying semantics** that other programming languages employ.

In order to mutate one object X into another object Y, all references to X must be redirected to Y. In Smalltalk this is achieved with the **become:** operation. This operation (see Figure 1) is best described as a **two-way become:** because references to X and Y are interchanged.

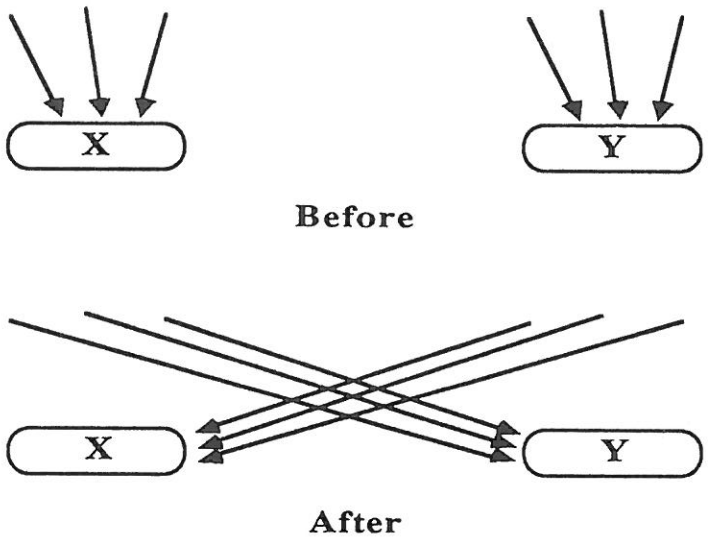


Figure 1

(X **twoWayBecome:** Y) The Two-Way **Become:** Operation

A variation of this operation that is not provided in Smalltalk is the **one-way become:**. In this version (see Figure 2), all references to X are redirected to Y (references to Y are unchanged).

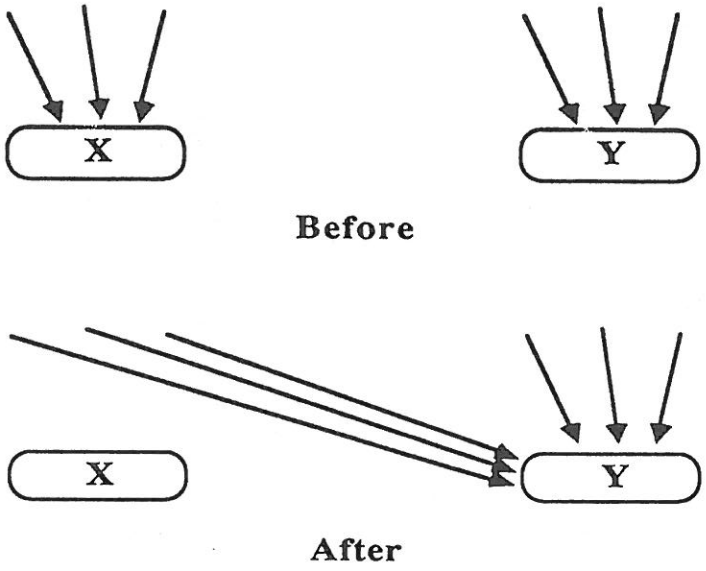


Figure 2

(X **oneWayBecome**: Y) The One-Way **Become**: Operation

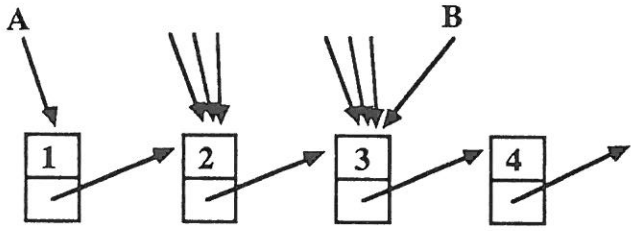
Neither of these operations can be simulated by the other. If we attempt to simulate a two-way **become**: with one-way **become**:, the first application of a one-way **become**: to say X forever loses all references to X. Conversely, to simulate a one-way **become**: with two-way **become**:, it is necessary to perform a union-style operation on B; i.e., pointers to some other object must be redirected to B without losing the existing ones. Redirecting references to B implies that B must be a parameter to a two-way **become**:. The unfortunate side-effect is that all existing references to B are moved. There is a third type of **become**:, an all-but-one one-way **become**: that can be used to implement both of the above types of **become**: non-primitively. It does however have the problem that the obvious implementation of a two-way **become**: is nonatomic.

Currently, the two-way **become**: is used primarily for handling object expansion. Such uses have the following flavor: (1) enter an operation like **grow**, (2) create a new object larger than the original O and store it in a local variable N, (3) make O **become**: N (afterwards N is bound to the original object), and finally (4) leave the **grow** operation (local variable N is discarded; hence, all references to the original object are lost). If a one-way **become**: were used instead, the only difference would be that N would remain bound to the new object. As an added bonus, using the one-way **become**: is faster than using a two-way **become**: since it involves only modifying the old object *in situ.*

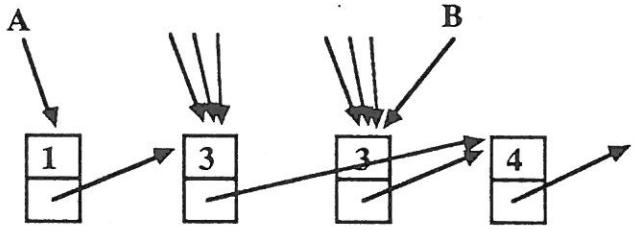
There are specialized uses however where a two-way **become**: is inadequate. A simple example is that of a sharable list with an operation such as **destructiveRemove**: with semantics that ensures the result is indistinguishable from the situation in which the element had never been inserted.

```
destructiveRemove: element
        self isEmpty ifTrue: [self error: 'element not found'].
        self first = element
                ifTrue: [self oneWayBecomes: self rest]
                ifFalse: [self rest destructiveRemove: element]
```

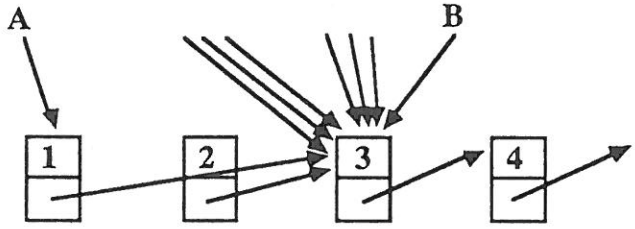
As can be seen from Figure 3, solutions for destructively removing element 2 without the one-way **become:** can be detected. If the element had never been there, A **rest** and B would be identical; i.e. "A **rest** == B" would return true. Even if we ignore such pedantic issues, the problem becomes more complex if the objects being linked are different types of objects. In that situation, the contents of one object might not fit in the other. Only the **become:** style solution works.



Before



After (Without Using Becomes: Operations)



After (Using A One-Way Becomes:)

Figure 3

Using The One-Way **become:** Operation To Implement **destructiveRemove**

All-But-One-One-Way become:

The **All-But-One One-Way become:** is an interesting operation because it can simulate both two-way and one-way **become:** operations. The all-but-one one-way **become:** operation is identical to the one-way **become:** except that the receiver object is returned by the primitive thus making it available to the sender of the **become:**. Simulating a one-way **become:** is trivial, (you need only to discard the result of the primitive), thus we will describe here how to do a two-way **become:** using only the all-but-one one-way **become:**. Note that this operation as shown *is not* done atomically.

twoWayBecomeNonPrimitively: newObject

```
" swap self and newObject non-primitively "

| oldSave newSave tempObjectOld tempObjectNew |

tempObjectOld = Holder new.
tempObjectNew = Holder new.
" first have all the referenents to self point to tempObjectOld. but keep one
 pointer to self oldSave "
oldSave = self allButOneWayBecome: tempObjectOld.
" have all the referenents to new point to tempObjectNew. but keep one
 pointer to newObject newSave "
newSave = newObject allButOneWayBecome: tempObjectNew.
" make all the pointers that used to point to self now point to the newObject "
tempObjectOld allButOneWayBecome: newSave.
" make all the pointers that used to point to newObject now point to self "
tempObjectNew allButOneWayBecome: oldSave
```

Efficient Implementation of become:

Initial Smalltalk implementations[Goldberg83] make use of a central object table which contained the real memory address of the object. In these systems **become:** is very inexpensive since only the object table entry needs to be updated. Recent implementations[ ] of large object space Smalltalks incur a major performance penalty for **become:**. In these systems every object is referenced directly via 32 bit pointer so that **become:** may require a complete sweep of memory to change all references to an object to the new object. This performance penalty has caused many implementors [Ungar86] to advocate the removal of the construct from the language. Indeed many implementations have rewritten their class libraries to eliminate or at least reduce the use of become.

Since we require a mutation operation to support object metamorphosis we cannot eliminate it, and in fact we want to make more use (disciplined) of it to support runtime changes of representation. One obvious approach is to mutate the object into a forwarding pointer to the new object. The problems with this solution are the cost of doing the message forwarding and the space wasted to hold the forwarding pointer. In Lisp systems where most objects are small(e.g. cons cells), the space penalty isn't significant, but for applications which make extensive use of large objects such as arrays the penalty is unacceptable.

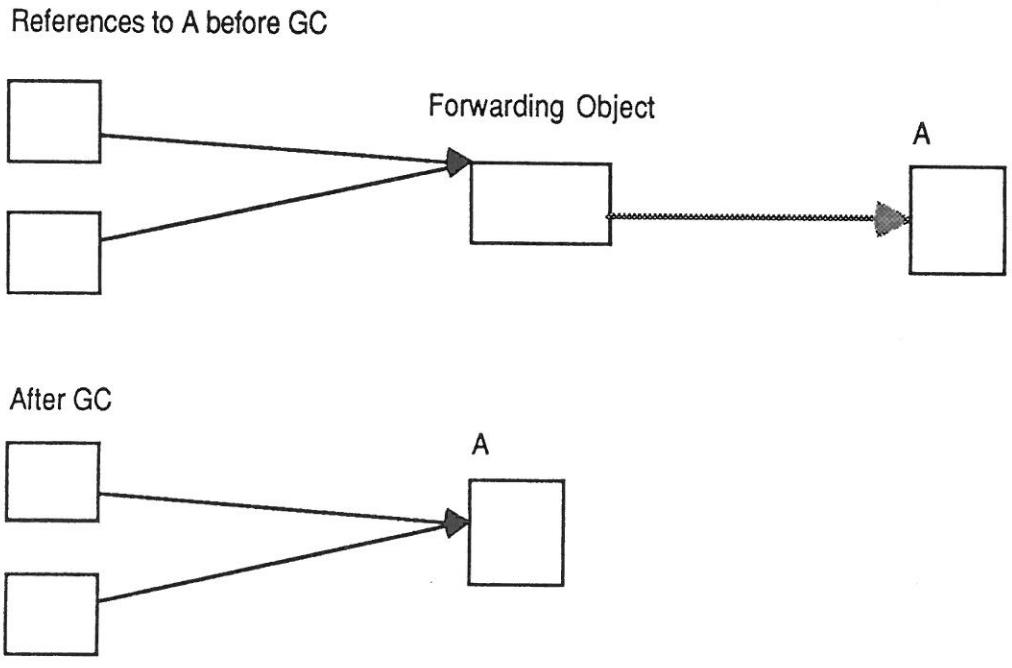


Figure 4 Garbage Collection Removes Forwarding Objects

We can solve both problems by using the garbage collector to short circuit the forwarding pointer. The application therefore only suffers the time/space penalty until a garbage collect, at which point both the pointer and its space are removed. Since most current implementations use a scavenging garbage collector, most forwarding pointers would exist for a very only a very brief time before been removed by the garbage collector. The process is illustrated in figure 4.

Efficient Implementation of Forwarding

Most current distributed smalltalks make use of some form of forwarding pointer to reference either data base objects or remote objects [Bennett, McCullough]. Smalltalk does not have any explicit notion of a forwarding pointer or indirection objects. User defined forwarding objects are implemented by defining a class of objects Forwarder which specialize the **doesNotUnderstand:** message. The **doesNotUnderstand:** message catches the message and forwards it to the associated object instance. Note the forwarding may also involve getting the object from a server or remote machine.

The two problems with this approach are the the time to look up the the **doesNotUnderstand:** message and the lack of primitive support. In order to send the **doesNotUnderstand:** message it is first necessary to try to locate the original message by searching the class structure. If method lookup fails, the virtual machine must then manufacture an appropriate **doesNotUnderstand:** message. The Smalltalk primitives do not expect forwarder objects and will fail. Some primitives operations such as == require a new interpretation in the context of distributed objects.

Suppose we add forwarding objects to the interpreter. If one makes forwarding pointers builtin objects, then they can be recognized at message send time and by the primitives. When the interpreter does a send, a test is made to see if it is a forwarding object and the message redirected to the real object. Initially it appears that a special case test in the send code is the way to speedup the forwarding operation. Unfortunately this test must be performed for every method send, degrading the performance of all message sends. If forwarding is performed automatically by the system, forwarders cannot have their own methods(such as distributed proxy objects require). Primitives must now check for the new builtin forwarding objects. Since primitives are very heavily used such checks have a negative impact on performance as a whole.

We require a solution where the cost of forwarding is small and only those who use it are required to pay for it. Instead of modifying the primitives in the virtual machine

we allow them to fail (this is done automatically in all smalltalks since all primitive parameters are type-checked) and check for forwarding in the failure code. This requires changing the Smalltalk code for handling failures, but requires no change to the underlying interpreter. Given that relatively few objects are actually forwarding pointers the cost of failing a primitive is minor relative to the cost of having an extra test in the primitive itself.

We use the method cache as the vehicle for resolving **doesNotUnderstand:** messages. Since the details of the cache implementation are important to understanding the message resolution speedup a brief description is provided here. The method cache is probed by using a function of the class of the receiver and the message selector to determine the cache entry to test. A simple effective strategy used in systems such as BS [Ungar86] is to exclusive-or the pointers and probe the cache at that value modulo the cache size. Strategies such as this have a 95% plus hit rate. In our system, when a forwarder receives a **doesNotUnderstand:** message, the method is entered into the method cache in the slot for the forwarders' class and the selector. (You can also enter the method in the cache slot for the real objects' class). The method entered consists of a change of self and a jump to the method of the real object. Once a message is placed in the cache it can be located as quickly as any other method making message forwarding only slightly more expensive than regular messaging. Our implementation has the advantage that regular message sends are not impacted and that the changes to the virtual machine are minimal.

Note that in order to correctly handle the cache lookup of forwarded messages we must have a separate forwarding class for each class of object that is to be forwarded. This is to insure that cache probes for the a selector can be differentiated for forwarders of different classes.

Special Cases ==

Many smalltalk systems use special bytecodes for often used messages, for example ==, that are implemented by inline execution of the method (no message is sent). The problem with this is that the == (and other such special send messages) can not be redefined. This presents a problem for instance when trying to test identity in the presence of forwarding objects, where identity is more complicated than testing for pointer equality. We do not attempt to define here what it means to be ==, instead we just state that redefinition of these special methods must be allowed.

Possible solutions to this problem include making == and the others *real* messages[Bennett87] and changing the implementation of == to use some form of object signature which is retained in the forwarding pointer or object[Bosworth87]. Both of these approaches are too slow. We prefer setting a bit flag (on a per class basis) in the object that indicates whether or not the message must be sent. The bit is set if the == (or other special selector) has been redefined in that class. The message is sent only if the class has the special selector methods redefined. This costs an extra test in every special no send bytecode. Given that most executions of == return true we can minimize the impact of the test by placing it in the failure path of the == byte code.

Conclusion

We have described why both object mutation and message forwarding are important and merit low level support in object oriented languages like Smalltalk. We have shown that both can be efficiently supported by relatively minor changes in the virtual machine and Smalltalk system. We note that such techniques are applicable to the implementation of new language notions such as delegation [Lieberman86] and laws, [Minsky87] both of which require the evaluation of the message as part of the lookup process. It remains and important and challenging problem to define the semantics and an associated efficient implementation of distributed sameness.

References

[Bennett87] John K. Bennet. *The Design and Implementation of a Distributed Smalltalk*. OOPSLA Proceedings Orlando, Florida, 1987.

[Goldberg83] Goldberg, A. and Robson, D. *Smalltalk-80: The language and its implementation*. Addison-Wesley, Reading, Mass. 1983.

[Krasner83] Krasner, G. *Smalltalk-80: Bits of history, words of advice*. Addison-Wesley, Reading, Mass. 1983.

[LTP86] LaLonde, W.R., Thomas, D.A., and Pugh, J.R. *An exemplar based Smalltalk*. First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon 1986.

[Lieberman86] Lieberman, H. *Using prototypical objects to implement shared behavior in object oriented systems*. First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon 1986.

[McCullough87] Paul L. McCullough. *Transparent Forwarding: First Steps*. OOPSLA Proceedings Orlando, Florida, 1987

[Minsky87] Naftaly H. Minsky, David Rozenshtein. *A Law-Based Approach to Object Oriented Programming*. OOPSLA Proceedings Orlando, Florida, 1987

[WirfsBrock86] WirfsBrock, A. and Caudil, P., *A Third Generation Smalltalk Interpreter*, First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon 1986.

[Ungar86] Ungar D., *SOAR - A Smalltalk without ByteCodes*, First Annual Object Oriented Programming Systems, Languages, and Applications Conference, Portland, Oregon 1986.

Carleton University, School of Computer Science
Bibliography of Technical Reports
Publications List (1985 -->)

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
KIS 5B6

SCS-TR-66 **On the Futility of Arbitrarily Increasing Memory Capabilities of Stochastic Learning Automata**
_____ B.J. Oommen, October 1984. Revised May 1985.

SCS-TR-67 **Heaps in Heaps**
_____ T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 **Partial Orders and Comparison Problems**
out-of-print M.D. Atkinson, November 1984. See Congressus Numerantium 47 ('86), 77-88

SCS-TR-69 **On the Expected Communication Complexity of Distributed Selection**
_____ N. Santoro, J.B. Sidney, S.J. Sidney, February 1985.

SCS-TR-70 **Features of Fifth Generation Languages: A Panoramic View**
_____ Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 **Actra: The Design of an Industrial Fifth Generation Smalltalk System**
_____ David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 **Minmaxheaps, Orderstatisticstrees and their Application to the Coursemarks Problem**
_____ M.D. Atkinson, J.-R. Sack, N. Santoro, T. Strothotte, March 1985.

SCS-TR-73 **Designing Communities of Data Types**
Wilf R. LaLonde, May 1985.
Replaced by SCS-TR-108

SCS-TR-74 **Absorbing and Ergodic Discretized Two Action Learning Automata**
out-of-print B. John Oommen, May 1985. See IEEE Trans. on Systems, Man and Cybernetics, March/April 1986, pp. 282-293.

SCS-TR-75 **Optimal Parallel Merging Without Memory Conflicts**
_____ Selim Akl and Nicola Santoro, May 1985

SCS-TR-76 **List Organizing Strategies Using Stochastic Move-to-Front and Stochastic Move-to-Rear Operations**
_____ B. John Oommen, May 1985.

SCS-TR-77 **Linearizing the Directory Growth in Order Preserving Extendible Hashing**
_____ E.J. Otoo, July 1985.

SCS-TR-78 **Improving Semijoin Evaluation in Distributed Query Processing**
_____ E.J. Otoo, N. Santoro, D. Rotem, July 1985.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-79 — **On the Problem of Translating an Elliptic Object Through a Workspace of Elliptic Obstacles**
B.J. Oommen, I. Reichstein, July 1985.

SCS-TR-80 — **Smalltalk - Discovering the System**
W. LaLonde, J. Pugh, D. Thomas, October 1985.

SCS-TR-81 — **A Learning Automation Solution to the Stochastic Minimum Spanning Circle Problem**
B.J. Oommen, October 1985.

SCS-TR-82 — **Separability of Sets of Polygons**
Frank Dehne, Jörg-R. Sack, October 1985.

SCS-TR-83 out-of-print — **Extensions of Partial Orders of Bounded Width**
M.D. Atkinson and H.W. Chang, November 1985. See Congressus Numerantium, Vol. 52 (May 1986), pp. 21-35.

SCS-TR-84 — **Deterministic Learning Automata Solutions to the Object Partitioning Problem**
B. John Oommen, D.C.Y. Ma, November 1985

SCS-TR-85 out-of-print — **Selecting Subsets of the Correct Density**
M.D. Atkinson, December 1985. To appear in Congressus Numerantium, Proceedings of the 1986 South-Eastern conference on Graph theory, combinatorics and Computing.

SCS-TR-86 — **Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacles Case**
B. J. Oommen, S.S. Iyengar, S.V.N. Rao, R.L. Kashyap, February 1986

SCS-TR-87 — **Breaking Symmetry in Synchronous Networks**
Greg N. Frederickson, Nicola Santoro, April 1986

SCS-TR-88 — **Data Structures and Data Types: An Object-Oriented Approach**
John R. Pugh, Wilf R. LaLonde and David A. Thomas, April 1986

SCS-TR-89 — **Ergodic Learning Automata Capable of Incorporating Apriori Information**
B. J. Oommen, May 1986

SCS-TR-90 — **Iterative Decomposition of Digital Systems and its Applications**
Vaclav Dvorak, May 1986.

SCS-TR-91 — **Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-92 — **ACTRA - A Multitasking/Multiprocessing Smalltalk**
David A. Thomas, Wilf R. LaLonde, and John R. Pugh, May 1986

SCS-TR-93 — **Why Exemplars are Better Than Classes**
Wilf R. LaLonde, May 1986

SCS-TR-94 — **An Exemplar Based Smalltalk**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-95 — **Recognition of Noisy Subsequences Using Constrained Edit Distances**
B. John Oommen, June 1986

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-96 **Guessing Games and Distributed Computations in Synchronous Networks**
J. van Leeuwen, N. Santoro, J. Urrutia and S. Zaks, June 1986.

SCS-TR-97 **Bit vs. Time Tradeoffs for Distributed Elections in Synchronous Rings**
M. Overmars and N. Santoro, June 1986.

SCS-TR-98 **Reduction Techniques for Distributed Selection**
N. Santoro and E. Suen, June 1986.

SCS-TR-99 **A Note on Lower Bounds for Min-Max Heaps**
A. Hasham and J.-R. Sack, June 1986.

SCS-TR-100 **Sums of Lexicographically Ordered Sets**
M.D. Atkinson, A. Negro, and N. Santoro, May 1987.

SCS-TR-102 **Computing on a Systolic Screen: Hulls, Contours, and Applications**
F. Dehne, J.-R. Sack and N. Santoro, October 1986.

SCS-TR-103 **Stochastic Automata Solutions to the Object Partitioning Problem**
B.J. Oommen and D.C.Y. Ma, November 1986.

SCS-TR-104 **Parallel Computational Geometry and Clustering Methods**
F. Dehne, December 1986.

SCS-TR-105 **On Adding *Constraint Accumulation* to Prolog**
Wilf R. LaLonde, January 1987.

SCS-TR-107 **On the Problem of Multiple Mobile Robots Cluttering a Workspace**
B. J. Oommen and I. Reichstein, January 1987.

SCS-TR-108 **Designing Families of Data Types Using Exemplars**
Wilf R. LaLonde, February 1987.

SCS-TR-109 **From Rings to Complete Graphs - Θ(n logn) to Θ(n) Distributed Leader Election**
Hagit Attiya, Nicola Santoro and Shmuel Zaks, March 1987.

SCS-TR-110 **A Transputer Based Adaptable Pipeline**
Anirban Basu, March 1987.

SCS-TR-111 **Impact of Prediction Accuracy on the Performance of a Pipeline Computer**
Anirban Basu, March 1987.

SCS-TR-112 **ε-Optimal Discretized Linear Reward-Penalty Learning Automata**
B.J. Oommen and J.P.R. Christensen, May 1987.

SCS-TR-113 **Angle Orders, Regular n-gon Orders and the Crossing Number of a Partial Order**
N. Santoro and J. Urrutia, June 1987.

SCS-TR-115 **Time is Not a Healer: Impossibility of Distributed Agreement in Synchronous Systems with Random Omissions**
N. Santoro, June 1987.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-116 **A Practical Algorithm for Boolean Matrix Multiplication**
M.D. Atkinson and N. Santoro, June 1987.

SCS-TR-117 **Recognizing Polygons, or How to Spy**
James A. Dean, Andrzej Lingas and Jörg-R. Sack, August 1987.

SCS-TR-118 **Stochastic Rendezvous Network Performance - Fast, First-Order Approximations**
J.E. Neilson, C.M. Woodside, J.W. Miernik, D.C. Petriu, August 1987.

SCS-TR-120 **Searching on Alphanumeric Keys Using Local Balanced Tree Hashing**
E.J. Otoo, August 1987.

SCS-TR-121 **An O(√n) Algorithm for the ECDF Searching Problem for Arbitrary Dimensions on a Mesh-of-Processors**
Frank Dehne and Ivan Stojmenovic, October 1987.

SCS-TR-122 **An Optimal Algorithm for Computing the Voronoi Diagram on a Cone**
Frank Dehne and Rolf Klein, November 1987.

SCS-TR-123 **Solving Visibility and Separability Problems on a Mesh-of-Processors**
Frank Dehne, November 1987.

SCS-TR-124 **Deterministic Optimal and Expedient Move-to-Rear List Organizing Strategies**
B.J. Oommen, E.R. Hansen and J.I. Munro, October 1987.

SCS-TR-125 **Trajectory Planning of Robot Manipulators in Noisy Workspaces Using Stochastic Automata**
B.J. Oommen, S. Sitharam Iyengar and Nicte Andrade, October 1987.

SCS-TR-126 **Adaptive Structuring of Binary Search Trees Using Conditional Rotations**
R.P. Cheetham, B.J. Oommen and D.T.H. Ng, October 1987.

SCS-TR-127 **On the Packet Complexity of Distributed Selection**
A. Negro, N.Santoro and J. Urrutia, November 1987.

SCS-TR-128 **Efficient Support for Object Mutation and Transparent Forwarding**
D.A. Thomas, W.R. LaLonde and J. Duimovich, November 1987.