Exploding Java Objects for Performance

Michael E. Noth

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2003

Program Authorized to Offer Degree: Computer Science and Engineering

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature Michael Mott

Date 16 Oct 2003

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Michael E. Noth

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

Alan Borning

Reading Committee:

David Notkin

Craig Chambers

Date: October 8, 2003

University of Washington

Abstract

Exploding Java Objects for Performance

by Michael E. Noth

Chair of Supervisory Committee:

Professor Alan Borning
Computer Science and Engineering

In the Java object-oriented programming language, a certain amount of memory overhead beyond an object's fields is required for each object instance. In simulations or other applications with millions of individual object instances, the per-object overhead can be substantial. This dissertation introduces the optimization technique of object exploding, in which fields of large numbers of objects are represented in parallel arrays. The problem space from which specific object exploding approaches are drawn is defined, and the "Restriction Approach" is described in detail. It is hypothesized that the restriction approach to object exploding can result in a factor-of-two improvement in application performance and a corresponding decrease in memory requirements. A prototype implementation is described, along with experimental results indicating that the restriction approach to object exploding can result in a factor-of-two or greater increase in application performance as well as a significant reduction in memory requirements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The author would like to express appreciation to the many people who contributed toward the success of this work. Many thanks to my advisor Alan Borning, who provided the right amount of guidance and direction while allowing enough latitude in my explorations. Thanks to Craig Chambers, David Notkin, the Cecil group, and the software engineering group at the University of Washington for providing invaluable feedback during numerous meetings and seminars. Many thanks to Paul Waddell and the UrbanSim group for allowing me to take part in design and development of UrbanSim. And finally, thanks to my other colleagues in the Computer Science department for providing an intellectually stimulating and fun environment for so many years.

This work has been supported in part by NSF Grant Numbers IIS-9975990, EIA-0090832, and EIA-0121326.

DEDICATION

To my parents Mike and Mary, who have provided all the love and encouragement any child could ever hope for.

Chapter 1

INTRODUCTION

The Java programming language is an object-oriented programming language in which all data entities are objects except for instances of the eight built-in primitive types. As is common in object-oriented languages, a certain amount of memory overhead beyond that required for the object's fields is required for each instance of an object. In certain domains, such as large-scale simulations of which the UrbanSim land use and transportation modeling system [41] is an example, very large numbers of objects are manipulated by the system. The memory overhead associated with each object can be quite significant, as high as dozens of bytes per object in some cases, and it adds up quickly in a system with many millions of objects. In addition, the layout of objects in memory often results in substantial performance penalties in the form of poor use of cache memory, particularly when only a small number of fields are used. Data prefetching may load many of an object instance's fields into cache, resulting in cache memory wasted by fields that are not used and by empty space resulting from word alignment of small fields.

Object exploding is a technique for increasing the time and space efficiency of applications written in an object-oriented language, particularly for those that that use large numbers (millions) of small objects, all of the same type. In this technique, exploded objects are represented by storing their fields in separate parallel arrays rather than using the conventional representation. This helps reduce wasted memory due to object overhead and alignment issues, and in some cases also improves cache locality and overall application performance. The primary goal of this work is to test the hypothesis that a Java source-to-source translation process to explode objects can result in significant (factor-of-two) improvements in both execution time and memory usage for a substantial real-world simulation application, such as the UrbanSim land use and transportation modeling system.

Chapter 2 presents the UrbanSim land use and transportation modeling system as a concrete example of the kind of application that benefits from the exploded object transformation process. Domains beyond UrbanSim that may benefit from the exploded object transformation process are described as well, along with a description of the exploded object transformation process and how it relates to other memory and performance optimizations applied to object-oriented languages. The chapter concludes with an example of a manual approach to the exploded object transformation process that illustrates its potential for saving memory and improving performance.

To help ground further presentation of the exploded object transformation, fundamental implementation concerns that affect any realization of the exploded object transformation are presented in Chapter 3. These fundamental implementation concerns act as decision points that define the overall problem space of approaches to the exploded object transformation problem. To help flesh out the problem space, four concrete implementations or proposals are described, including the Restriction Approach which is the focus of this work.

The Restriction Approach is fully introduced in Chapter 4 through a presentation of its goals, the restrictions it operates under which give it its name, and the features it provides. Semantics of the Restriction Approach and the implementation strategy used to realize the approach are presented in Chapters 5 and 6.

The template-based approach to specialized collections for exploded objects as required by the Restriction Approach and provided by the prototype implementation is described in Chapter 7. Evaluations of the prototype implementation of the Restriction Approach are found in Chapter 8, where results demonstrate that, at least under certain low-memory conditions, the Restriction Approach can provide substantial performance benefits over the normal Java `Object` version of an application, and generally uses much less than half the memory of the Java `Object` version.

Chapters 9 and 10 contain related work and possible future extensions relating to the Restriction Approach and its prototype implementation. A summary of the Restriction Approach and other conclusions are found in Chapter 11. Appendices A and B provide implementation details relating to the prototype implementation of the Restriction Approach as well as usage information for the prototype implementation.

Chapter 2

MOTIVATION

There are many application domains that contain large numbers of generally small objects that can benefit from the exploded object transformation process. For the purposes of this work the UrbanSim simulation system is the primary motivating application domain, though by no means the only application domain of interest. The UrbanSim simulation environment is described in some detail in the following sections, to provide a concrete example of an application that can benefit from exploded objects. Other domains are described in subsequent sections. This chapter concludes with a description of an early application of exploded objects to UrbanSim and the results, placing the exploded object approach in a broader context, as well providing an illustration of the potential benefits of the exploded object transformation by showing what effects a simple version of the transformation have on a test problem.

2.1 UrbanSim

The UrbanSim land use and transportation modeling system [32, 41] is a large-scale, fine-grained simulation system implemented in Java. Simulation entities such as households, jobs, land parcels, grid cells containing environmental information, etc., are represented individually within an object store. UrbanSim consists of a collection of model components that operate on the simulation entities, such as a residential location choice model component that determines where a given household may choose to locate, a developer model component that simulates what real estate developers may choose to do with vacant or redevelopable land, a travel model that determines accessibility values between different traffic zones based on the transportation network, and so on.

The primary goal of UrbanSim is to provide a useful system for use in planning by municipal planning organizations, concerned citizen interest groups, and policy experts. Their interests typically involve evaluating changes in tax structures, zoning laws, modifications to the urban growth

boundary of a city, alterations to the transportation network such as the installation of light rail, impact of environmental regulations, etc. UrbanSim provides for the creation of scenarios incorporating the changes then generates "possible futures" based on simulation execution.

To help satisfy its primary goal, UrbanSim also serves as an experimental testbed. Urban planners, environmental modelers, and other related researchers use UrbanSim as a platform in which they can easily deploy alternative versions of existing model components or introduce entirely new ones. Modification and replacement of model components, and extension of UrbanSim through the introduction of new model components, allows researchers to experiment with different modeling styles, evaluate the effects of different spatial and temporal granularities, and perform interdisciplinary experiments involving interactions and feedback loops from processes in a range of domains, including travel modeling, land development, air and water pollution, weather, etc.

To further its usefulness to a wide range of users, UrbanSim is designed as an open and transparent system, where as many aspects of the simulation and its components as feasible are made clear and all decisions made explicit.

To be of use for both experimentation and actual deployment to simulate and model specific real-world urban areas, UrbanSim must be able to accommodate a wide range of input data sizes. For example, the original experimental testbed dataset for Eugene, Oregon contained on the order of one hundred thousand total simulation entities (including households, parcels, businesses, etc). UrbanSim has been deployed in Honolulu and Salt Lake City, where the latter has on the order of a million total simulation entities. Future deployments may include cities such as Paris and its surroundings or Los Angeles, which would involve another order of magnitude increase in the number of simulation entities. It may be of interest to attempt to use UrbanSim to simulate even larger areas, such as collections of interacting cities as found in Japan and the northeastern United States, which would likely involve yet another order of magnitude increase. Clearly, it is important to be able to represent very large numbers of objects within UrbanSim to allow it to be of use for a range of simulations, and it is highly desirable for a single design of the system to be extensible to meet these and other demands.

2.2 Other Application Domains

There are many other domains that use or could benefit from object-oriented programming and design techniques that also make use of patterns similar to those found in UrbanSim. Potential benefits from the application of exploded objects as described in this work would arise through being able to represent larger numbers of simulation entities more efficiently, or to handle larger data sets, reducing the computational requirements to improve simulation usage for experimental purposes, or making the simulation more accessible by reducing memory and/or computational requirements. Several of these domains are described below.

2.2.1 Agent-Oriented Simulations

Agent-oriented simulations are simulations in which individual actors, or agents, are the key driving force of a simulation. Agents typically interact with each other and their environment by gathering information about their surroundings, communicating with other agents, and acting based on their observations and internal state. Three areas of agent-oriented simulation that typically have large numbers of agents, making them more suited to the exploded object transformation, include agent-oriented simulations in the social sciences, economic simulations, and general-purpose agent-oriented simulation frameworks.

Many areas within the social sciences have begun to use agent-oriented simulations for research purposes [17]. Such simulations typically involve some number of agent objects that observe the simulated world around them through sensors of one form or another, and then act upon the world through actuators. Examples include Sugarscape [14], in which agents gather resources, trade them, form groups, etc., and ongoing work on using simulation modeling for anthropologists such as simulating changing kinship relations over time, village migration, etc. [15].

Agent-oriented economic simulations have increased in sophistication and scope in recent years, as demonstrated by the emerging discipline of Agent-Based Computational Economics (ACE) [38]. ACE entails studying and constructing autonomous agents whose repeated local interactions create global regularities, such as trade networks, market protocols, etc.

There are a large number of other simulations in the economic domain that could benefit from increased performance and/or scalability due to the use of exploded objects. These include the As-

pen project [35], which intends to perform an agent-oriented microsimulation of the entire United States economy, agent-oriented equilibrium-based market simulations [40], and agent-based dynamic manufacturing scheduling [37], to name a few.

Several frameworks have been proposed for specifying and executing agent-oriented simulations. Many of the frameworks are hierarchical and explicitly object-oriented in nature, and as such, could benefit from exploded objects to improve performance and assist with simulations involving very large numbers of agents. These frameworks include Swarm [30], which is implemented in Objective-C as well as Java, the High-Level Architecture [11], which provides a framework in which multiple computation entities rely on an explicit message-passing infrastructure to facilitate large-scale distributed simulations, and the proposed hierarchical agent simulation frameworks of MASSIF [28] and MIMOSE [29].

2.2.2 Environmental Modeling

Environmental modeling encompasses a broad range of models, such as modeling air or water quality, land cover changes over time, weather, etc. Many environmental models are based on cellular automata, where the simulation environment is divided into a number of grid cells and simple, generally homogenous computations are performed on each that typically involve only local properties. These models are frequently integrated with geographical information systems (GIS) to aid in visualization and analysis, such as FIREGIS [18] which is used to simulate the spread and evolution of forest fires.

Other environmental models, such as the Patuxent Landscape Model [10], are based on a collection of finite elements that break apart the simulation environment into discrete spatial units, but unlike cellular automata models, they may perform sophisticated computations involving local and global characteristics, temporally-based event simulations, etc. Both the Patuxent and FIREGIS style of models may benefit from the use of exploded objects to allow "real" objects (from the programmer's standpoint) to represent grid cells or other spatial units, and thus leverage more object-oriented designs and the attendant benefits, as opposed to the array-based structures that are more commonly used in implementations.

2.2.3 *Transportation Modeling*

Transportation modeling is another domain in which exploded objects may be of benefit. In particular, a more recent emphasis on activity-based travel models [27], wherein individuals or households plan trips based on a daily or periodic activity schedule, and those trips are then scheduled onto the transportation network of roads, public transit, pedestrian travel, etc., may tend toward having larger numbers of simulation entities. As the number of entities increases, the potential memory savings and performance improvement increase as well.

2.2.4 *Numerically-Intensive Scientific Models*

There are several aspects of Java that limit its ability to be deployed for high-performance, numerically-intensive scientific modeling [21]. The use of exploded objects can help alleviate several of these limitations, namely providing for lightweight (or "lighterweight") classes, and the ability to use this mechanism to allow for very large numbers of small class instances, such as for complex numbers or packed arrays.

2.3 *Exploded Objects in UrbanSim*

In the original version of UrbanSim, simulation entities were directly represented by Java objects (subclasses of the `java.Object` class). As UrbanSim began to be tested with larger and larger data sets, it became clear that memory usage was growing prohibitively large and that the rate of growth would preclude using that version of UrbanSim for all but the smallest data sets. A simple analysis[1] of memory usage revealed that the Java object overhead in the Java virtual machine (JVM) used at the time was on the order of 16 to 20 bytes, and that alignment of object fields on word boundaries by the JVM was resulting in substantial wasted memory as many fields of UrbanSim objects are very small (e.g., single **`byte`**s). Figure 2.1 shows the fields present in a typical `Household` object for UrbanSim, along with the typical in-memory layout of a single `Household` instance that shows how much of the memory required by one instance may be overhead. For larger objects, the amount

[1]To estimate the memory being used per object instance, the total size of the heap was examined before and after allocating a large number of object instances in an array, with the difference in memory usage being divided by the total number of allocated objects. Garbage collection was forced prior to examining the heap size in both cases.

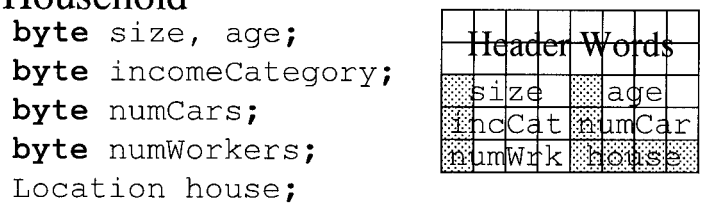

Figure 2.1: Fields and memory layout of a typical `Household` object, showing how much memory may be lost due to per-instance overhead. In the figure on the right, shaded boxes are those required to store the fields as declared in the `Household` definition, with empty boxes showing overhead. One small box represents one byte of memory.

of overhead may vary. If small fields are word-aligned, then each additional small field will add several bytes of additional overhead. If small fields can be packed together into words, then larger numbers of small fields may reduce the absolute amount of overhead; for example, if a larger object has a number of **`byte`** fields that is divisible by four, and **`byte`** fields are packed together into 32-bit words, then no alignment-based overhead will be present in the larger object as compared to a smaller object with less than four single-**`byte`** fields. In most cases, larger objects will tend to have less per-instance overhead on a proportional basis simply by virtue of having more data associated with them.

To attempt to alleviate the memory concerns, the UrbanSim object store was redesigned to store simulation objects in a series of parallel arrays instead of storing them as traditional Java objects. This original form of manually-exploded objects[2] was complex, error-prone, and difficult for clients to use, but resulted in a memory reduction of roughly a factor of seven, as well as some performance improvement. This experience serves as one data point that object exploding may result in substantial memory savings for large-scale simulations and other applications dealing with very large numbers of individual object instances. Figure 2.2 shows the object layout resulting from the conversion to exploded objects. Note that virtually all per-instance overhead has been removed, as small fields are packed into arrays to avoid overhead from word alignment and header words are

[2]The term *object exploding* is used to differentiate this general problem from the software-engineering problem of object slicing [25] where objects and their contexts are used to generate program slices containing statements and values that may affect a program value at a given program point.

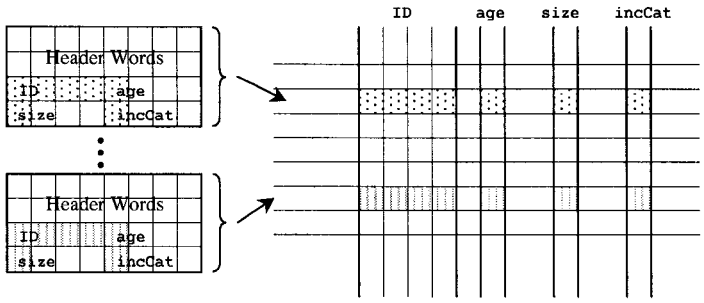

Figure 2.2: An instance of an exploded object type is represented as a "slice" through the set of parallel arrays that store its fields.

entirely eliminated.

Other literature [7, 12, 23] demonstrates how reorganizing data layout of individual objects or collections of related objects can result in substantial performance gains through improving cache locality. Other techniques applicable to object-oriented languages improve performance by reducing memory usage through specialized allocation [20], or elimination of object header words or other overhead [1, 4, 3, 6]. Object exploding can achieve many of these benefits as well, as suggested by the implementation strategy presented in Chapter 6 and by the experimental results, including microbenchmarks, found in Chapter 8.

Many applications, particularly in simulation domains of primary concern for the purposes of this work, tend to access a small number of fields of a large number of objects. The object layout resulting from the exploded object transformation process is more cache-friendly than traditional object layouts. In traditional object layouts, where all fields of one instance of an object are sequential in memory, accessing one field will cause many other fields to be loaded into the data cache through pre-fetching. If the application then uses only a small number of other fields, most of the data in the cache will be unused, and therefore cache memory will be wasted. Under the exploded object representation, where one field for all instances are stored sequentially, accessing one field will result in the same field for many other instances being loaded into the data cache. As a result, subsequent similar computations involving other instances will be more likely to involve cached data.

UrbanSim in particular is a prime example of an application that can benefit from this prop-

erty of the exploded object data layout, as most of its core computations consist of iterating over significant subsets of simulation entities of a given type and performing an identical computation on each instance, accessing a small number of its fields in the process. Other domains, such as finite element systems, agent-oriented simulations, and finite automata, often contain analogous core computations.

The exploded object transformation process provides for the opportunity to eliminate all of the header words normally found in Java virtual machine implementations by requiring only storage for actual instance fields. While additional storage may be required to support certain Java language features such as synchronization or run-time type information for use in dynamic dispatching, exploded objects provide a range of choices and possibilities for optimizations that are not normally present in Java.

Exploded objects also permit custom allocation strategies to be used with exploded object instances, allowing allocation tuning to address specific application requirements in ways that would otherwise require a customized Java virtual machine. Customized allocation strategies can significantly reduce the computational costs associated with object allocation.

2.4 Motivating Example: Manually-Exploded Objects

To demonstrate the potential for memory savings and performance benefits from the exploded object transformation process, consider a simple case of manually-exploded objects. For this example, no attempt is made to preserve normal Java object semantics; client code must be rewritten to make use of this type of manually-exploded object.

As a test, a simple Household class object is defined. It contains six `byte` fields, one for income category, age, size, number of cars, children, and workers, a `boolean` flag indicating if the household owns its home, and two `int` values storing the ID of its location and its own internal ID. The total size of the object's fields is 15 bytes, assuming one byte to store a `boolean`. Figure 2.3 shows the class definition used for the manual test. The total time required to create five million of these objects in one array is measured, followed by measuring the total time required to access one field (the income category) in each instance, in the order in which they were created. This access pattern is quite common among UrbanSim-style simulations, where some component of the simu-

lation iterates over all objects and retrieves a small number of fields; aggregation, averaging, and minimum/maximum determination are all examples.

A very crude version of the object is defined in exploded form. Each field of the object has been stored in an array of the field's type, resulting in a set of parallel arrays as shown in Figure 2.4. A slice through the arrays, defined by an identical index value, represents an object. The total time required to create five million of these manually-exploded "objects" is measured, followed by the time required to access one field in each object in the order in which they were created.

For both cases, memory measurements are taken, using the `java.Runtime` class's `freeMemory` and `totalMemory` methods. Heap size for the virtual machine is pre-set at 256MB. When runs were made with Sun's JDK 1.4.1_01-b01, with the Java HotSpot(TM) Client VM, build 1.4.1_b01, running in mixed mode, the results were promising, as show in Figure 2.5.

The manually-exploded objects occupy roughly sixty percent of the memory required by the Java-style objects. Creating the exploded objects is faster by a factor of nearly four, and accessing one field in each is faster by a factor of two and a half. The Java-style objects have a per-object memory overhead of 9 bytes in the VM that was used. Assuming 32-bit word alignment and that the VM performs optimal field packing to minimize wasted space due to data alignment[3], one byte was wasted due to aligning the seven one-byte fields. The remaining 8 bytes are likely used to store the virtual function table pointer, a precomputed hash code, a lock, or other data.

Clearly, this experiment is somewhat contrived, as only a single field of the object is accessed, no other computation is performed in the iteration loop, and no attempt has been made to preserve Java-style object semantics, in that exploded object "instances" cannot be the target of method invocation, references to them cannot be passed to methods in the normal Java fashion without significant additional work, and generally exploded object "instances" and references to them are handled by both developers and the run-time system in fashions wholly different from normal Java objects. However, this simple experiment, along with results from the original UrbanSim application, have shown that transforming an application to use exploded objects can result in substantial benefits. More detailed and realistic experiments are provided in Chapter 8.

---

[3]Note that as recently as version 1.3.1, Sun's JDK did not perform field packing. Under the versions of the JDK used to develop early versions of UrbanSim, the wasted memory was often several times greater than the total size of all object fields.

```
1 public class Household {
2
3   public int hhID;
4   public static int nextHHID = 0;
5
6   // Income category of the household
7   public byte incCat;
8   // Household's location, -1 for none
9   public int hhLoc;
10  // Age category of the household
11  public byte age;
12  // Flag indicating if the household owns its home
13  public boolean ownsHome;
14  // Size of the household
15  public byte size;
16
17  // Number of cars, workers, and children
18  public byte numCar;
19  public byte numWrk;
20  public byte numKid;
21
22  public Household() {
23    hhID = Household.nextHHID++;
24    ...
25  }
```

Figure 2.3: Simple Household class definition used for manual object-exploding example.

```
1 public class Household_FieldArrs {
2
3   public int[] hhID;
4   public static int nextHHID = 0;
5
6   // Income category of the household
7   public byte[] incCat;
8   // Household's location, -1 for none
9   public int[] hhLoc;
10  // Age category of the household
11  public byte[] age;
12  // Flag indicating if the household owns its home
13  public boolean[] ownsHome;
14  // Size of the household
15  public byte[] size;
16
17  // Number of cars, workers, and children
18  public byte[] numCar, numWrk, numKid;
19
20  // Method to create a new "instance"
21  public int create() {
22    int idx = nextFreeIdx++;
23    hhID[idx] = Household_FieldArrs.nextHHID++;
24    ...
25    return idx;
26  }
```

Figure 2.4: Manual conversion of simple Household class used for manual exploded-object example.

Original Java version:

```
All timings (ms):
create: 4969
printIncCatRange: 531

Memory used: 114 MB
```

Manually-translated version using exploded objects:

```
All timings (ms):
create: 1312
printIncCatRange: 204

Memory used: 71 MB
```

Figure 2.5: Timing and memory results from manually-exploded example.

Chapter 3

CHARACTERIZATION OF THE EXPLODED OBJECT PROBLEM SPACE

There are many different techniques and implementation strategies that can be used to realize the exploded object transformation. The manual conversion example presented at the end of the previous chapter is one such approach. This chapter describes the problem space from which approaches to the exploded object transformation are drawn. The "Ideal" approach is presented first, along with implementation concerns that make the Ideal approach impractical for the purposes of this work. Five dimensions along which the Ideal approach could be modified to make it more practical are then presented, namely syntax, semantics, compiler modifications and requirements, run-time requirements, and environment considerations. These dimensions serve to define the problem space from which approaches to the exploded object transformation are drawn. To help illustrate the interactions between decisions along each dimension, three concrete examples of approaches are presented at the end of the chapter.

3.1 The Ideal Approach

For purposes of illustration, consider a proposed "Ideal" approach to the exploded object transformation problem. In the Ideal approach, the only difference between exploded objects and non-exploded objects is the internal object layout, with accompanying memory savings and performance benefits for the exploded object version. From the programmer's perspective, exploded and Java objects are identical, except that exploded object types are explicitly tagged as such through the addition of an "`exploded`" keyword. Exploded and Java objects can be freely mixed, including through inheritance, subsumption, and interface implementation, garbage collection of exploded object instances is the same as for Java `Object` instances, and no burden is placed on the developers of the system to make use of exploded objects.

Many implementation concerns would need to be resolved in an implementation of the Ideal

approach. Implementation concerns are important to take into account, as different fundamental implementation decisions and concerns dramatically affect what features can be provided, what limitations are imposed, and what tradeoffs involving performance and development complexity or feasibility are available.

An implementation of the Ideal approach would require substantial modifications to the Java virtual machine in order to be practical, for reasons described below. Modifying the JVM is generally to be avoided, as doing so eliminates one of Java's largest advantages, namely portability, as a version of the modified VM must be made available for each target platform. In addition, it may be difficult to take advantage of future developments in VM technology if one has a custom JVM, as any future developments must be integrated with the changed virtual machine instead of merely switching to make use of the latest version of the JVM as supplied by others. It is possible to make modifications to the Ideal approach, by compromising on certain areas, to end up with an approach that retains many of its advantages but will run on any Java virtual machine.

An implementation of the Ideal approach would require significant amounts of support. At a minimum, extensive compiler support would be required to automatically perform the exploded object transformation. To provide for any degree of efficiency in internal representations of exploded objects that can be interchanged with Java objects, and to allow for garbage collection, modifications to the JVM would be required.

To see why this is the case, consider the difficulties inherent in allowing an exploded object to be used where a Java `Object` is expected, while restricting an implementation to generate normal Java source or bytecode that can run on any Java virtual machine. Even if one makes the somewhat unrealistic assumption that the compiler can perform perfectly precise alias analysis to exactly determine when any particular exploded object instance will be used where a Java `Object` subclass is expected, the compiler would have to either wrap the exploded object instance with a Java `Object`, build a distinct Java `Object` instance that wholly replaces the exploded object instance and destroy the exploded object instance in the process, or provide specialized data structures to support Java `Object`-equivalent operations on an exploded object, such as storage in a normal Java collection.

Any choices which involve converting between or wrapping exploded object instances with normal Java `Object` subclasses will suffer performance penalties due to the cost of conversions, and may partially or wholly negate the benefits of the exploded object transformation if the system

ends up creating one Java `Object` instance per exploded object instance.

Note that in some cases, wrapping an exploded object instance with a Java `Object` wrapper is inadequate. For example, synchronization involving the wrapper object does not prevent asynchronous access to the underlying exploded object instance being wrapped, but only prevents manipulations directly involving the wrapper object. If multiple wrapper objects are present, such as a persistent reference to one being stored and used later, or if there are any other mechanisms for interacting with exploded objects apart from Java wrappers, then synchronizing on a wrapper object provides no guarantees about concurrent operations on the exploded object instance. Likewise, alternate data structures are inadequate for other tasks, such as reflection, as the Java reflection interface requires access to classes created by the Java virtual machine and that cannot be created by user-level code as there are no public constructors of the classes in the `java.lang.reflect` package. Explicitly replacing an exploded object instance with an "equivalent" Java object has limitations as well, in that doing so will impose serious performance penalties due to object creation and deletion, requires additional infrastructure to ensure that there are no dangling references to the destroyed exploded object instance, where such infrastructure may be extremely difficult to provide at the user-code level, and may raise other issues involving object identity between exploded and non-exploded instances of the "same" type.

Once it has been established that the Ideal approach requires a customized Java virtual machine, it becomes somewhat easier to make choices regarding other implementation concerns. The desire to support inheritance between and subsumption involving Java and exploded object types, as well as inheritance and subsumption involving different exploded object types, is feasible when one has chosen to customize the Java virtual machine. Likewise, supporting automatic garbage collection of exploded object instances is feasible through VM modifications.

There are several different modifications to the Ideal approach that one could perform that would affect its practicality. If one decided to require explicit deallocation of exploded object instances instead of providing for automatic garbage collection, modifications to the virtual machine could be simplified; only those modifications needed to provide efficient support for typing issues would be required.

A more significant modification would be to disallow modifications of the Java virtual machine, and instead rely solely on compiler and language modifications. It would be extremely difficult

to provide efficient support for subsumption between exploded and Java objects, or to allow for automatic garbage collection of exploded object instances without VM modifications, for reasons described previously. Instead, one could impose restrictions on interactions between Java and exploded object types, such as wholly disallowing interactions by creating independent type systems for each kind of type, and assume explicit deallocation of exploded object instances instead of garbage collection. It would be necessary to restrict or eliminate support for synchronization or reflection involving exploded objects, as well, for reasons described previously.

The implementation concerns and related issues expressed above can be divided into five categories for purposes of defining the problem space of exploded object transformations. These categories are syntax, semantics, compiler requirements, run-time environment requirements, and other environment-related concerns. These categories of concerns are not wholly independent, but their relationships serve to further characterize the space by demonstrating what regions of the space are practical or impractical. Each of these categories is described in more detail in the subsequent sections.

3.2 Syntactical Considerations

Syntactical considerations can be divided into two categories: those involved in definitions of exploded object types, and those involved in usage of exploded objects. In the Ideal approach, the only syntactic changes involve the addition of the `exploded` keyword, used with class definitions to indicate that the class is to be converted into exploded form internally. When considering modifications to the Ideal approach, there are a range of different options for syntax used in definitions, including no special syntax at all, simple tags that mark a type as being exploded, and complex specifications that provide instructions on internal layout. If no syntax is used to define or tag object types as being exploded, then the compiler and/or run-time infrastructure must be able to automatically determine how each object type should be represented. Compiler-level determination is likely to be impractical unless the compiler can perform semantic analysis of program code to determine which usage patterns may be fruitful for the exploded object conversion process; alternatively, if semantics of exploded objects are identical to normal Java objects, or differ in well-defined fashions, the compiler could choose to automatically tag all types which can be converted legally. Similar

determinations could be made at run-time, if the run-time infrastructure is robust enough to support converting between exploded and non-exploded types in a running system, though such conversion and analysis would need to be efficient to avoid the cost of analyses and conversion overwhelming the benefits.

Tags used to annotate class definitions to indicate that they should be converted to exploded types are another option. Tags range from the very simple, such as an `exploded` tag associated with a class definition, to more complicated options that may indicate that conversion to exploded form is always required, is desired but not required, is to be based on its usage elsewhere either statically or dynamically, etc. Simpler tags, particularly those that indicate unconditional conversion, are likely to require only compiler-level support, whereas conditional tags may require substantial analysis at compile-time and/or run-time, depending on the conditions. For example, a tag indicating that the type should be converted if its usage obeys any static type-based restrictions could be analyzed by the compiler, but a tag indicating that only types created in sufficient quantities should be converted may require run-time analysis or profile information.

A more sophisticated option beyond tags is the use of detailed specifiers that indicate exactly how an object's fields are to be positioned and grouped in memory. Specifiers could be as simple as hints indicating that certain fields are to be grouped together or not, or as complex as a domain-specific language that indicates exactly which fields of which object types should be grouped and how the grouping should be organized. See Keunwoo Lee's qualifying exam write-up [24] for an example of such a domain-specific language.

With respect to syntactic considerations involving usage of exploded objects, options include no special syntax so that exploded objects are syntactically identical to Java objects, or special syntax used to create, delete, access, and/or modify instances. If exploded objects can be used in an identical or largely identical fashion to Java objects, then identical syntax can be used. If there are differences in usage, such as special syntax required to create instances, or if exploded object instances must be explicitly deleted, then new or modified syntax with respect to Java can be used. In cases where identical syntax can be used, it may be desirable to use different syntax to differentiate exploded and non-exploded instances. For example, different syntax could be used to provide a cue to developers that they should use different data types for storing collections of exploded objects for efficiency reasons.

In the compromise approach taken in this work, namely the Restriction Approach as described in later chapters, the only syntactic modifications are the same as for the Ideal approach. The **`exploded`** tag has been added for use with class definitions to indicate that they are to be converted to the exploded form.

3.3 Semantic Considerations

The second category of fundamental implementation concerns involve semantics, which is dominated by typing issues. The two main choices involving typing issues are determining what typing relationships are allowed between exploded and non-exploded types, and what typing relationships are allowed between different exploded types. The degree of support for interface implementation is another significant choice.

In the Ideal approach, exploded object types are treated the same as non-exploded object types, in that the two kinds of types can be freely mixed, there are no limitations on inheritance or subsumption between the kinds, etc. As described below, however, it is not practical to support this degree of mixing in an efficient manner without modifying the virtual machine.

With respect to typing relations between exploded and non-exploded types, the key choices are whether to allow any relations to exist at all, whether exploded types can inherit from non-exploded types or vice versa, or whether subsumption between exploded and non-exploded types is allowed. If no typing relations are allowed, then exploded and non-exploded types are wholly independent, and there are no particular ramifications in terms of required infrastructure such as library, compiler, or VM modifications. However, disallowing any typing relations limits the ability to reuse code through inheritance, and can complicate the adoption of exploded objects in an application by mandating what may be non-trivial changes if now-exploded object types were stored with non-exploded types in collections or mixed other ways.

Choosing to allow inheritance between exploded and non-exploded types without allowing subsumption is relatively straightforward to provide, as inheritance can be performed using code replication. However, if subsumption is not allowed, so exploded object types that inherited from non-exploded object types may *not* be used where the original non-exploded type is expected, the resulting system may no longer have normal Java `Object` semantics, in which inheritance implies

subsumption. In particular, it would be undesirable for a system to allow inheritance between exploded and non-exploded types in both directions without also supporting subsumption, as one could write an application where a non-exploded type C inherits from an exploded type B which in turn inherits from another non-exploded type A. According to normal Java `Object` semantics that require support for subsumption, an object of type C should be usable wherever an object of type A is expected. In a system that supports inheritance but not subsumption, it would no longer be legal to use an object of type C where an object of type A is expected due to the inheritance chain tracing through the exploded type B, and this change in semantics could result in confusing errors.

A more general approach is to support both inheritance between and subsumption involving exploded and non-exploded types. While doing so would avoid the potentially strange errors as described above, it may be very difficult to support subsumption between exploded and non-exploded types without requiring a customized virtual machine as explained in Section 3.1.

If one chooses to disallow subsumption between exploded and non-exploded types, it may still be possible and useful to allow inheritance and subsumption between different exploded object types. Supporting inheritance between different exploded object types can be done through code replication without requiring a customized virtual machine. Supporting subsumption between different exploded object types is somewhat trickier, but doable without requiring modifications to the JVM. Run-time type information can be associated with each exploded object reference within user-level code, and that information can be used to simulate dynamic dispatching by examining the run-time type and explicitly invoking the appropriate method; this can all be done through user-level code such as a **`switch`** statement, but may have performance implications as compared to the internal mechanisms for a normal Java virtual function call.

A somewhat related typing issue involves implementation of interfaces. Choices include disallowing any interface implementation by exploded object types, forbidding implementation of the same interface by both an exploded and non-exploded type, allowing any implementation of interfaces, and choosing whether or not exploded object instances that implement interfaces can be cast to the interface types.

If no interface implementation is permitted by any exploded object type, no additional infrastructure is required. If compiler modifications are allowable, it is a simple local check to statically verify that this restriction is obeyed, by ensuring that no exploded object type has an **`implements`**

clause. A local check no longer suffices if exploded object types can inherit from non-exploded types, as the non-exploded type or its supertype(s) may have implemented interfaces, but such a combination of choices does not make sense.

If interfaces may be implemented by both exploded and non-exploded types, but no interface may be implemented by both kinds of type, then whole-program analysis is required to verify that no interface implemented by any non-exploded type is also implemented by any exploded type, and vice versa. Under this choice, one might wish to have explicitly-declared "exploded interfaces" that can only be implemented by exploded object types; such a decision would allow for strictly local analyses as opposed to global analyses that would be required if any interface might be implemented by either kind of type. Allowing "exploded interfaces" or other implementation of interfaces by exploded object types allows exploded object semantics to more closely resemble those of normal Java objects, which may be desirable from programmer familiarity and usability standpoints.

If interfaces may be implemented by both exploded and non-exploded types, in any combination, then the approach must also support subsumption between exploded and non-exploded types, as an exploded type could be cast to an interface type, and that interface type used as a normal Java type. Alternately, unrestricted interface implementation could be permitted, but explicit or implicit casting of exploded objects to interface types could be forbidden, which would avoid requiring support for subsumption between Java and exploded object types, but the restriction on casting may be unpleasant for programmers trying to apply exploded objects to an application that was not written with that restriction in mind.

If there are distinct interfaces implemented by exploded and non-exploded object types, such as "exploded interfaces", it is easier to support casting of exploded object types to interface types. As long as exploded interface types are distinct from normal Java interface types, it should be possible to associate run-time type information with each instance of an exploded interface type and then perform dynamic dispatching through user-level code or a customized JVM in much the same way such dispatching is performed for normal Java interface types.

3.4 Compiler Requirements and Concerns

The requirements imposed on the compiler by a given approach are another dimension along which different approaches can be differentiated. Options for compiler requirements include no direct compiler support, some compiler support but no language-level support, or combined compiler and language- level support. Clearly, the Ideal approach would require substantial compiler-level support, along with at least minimal language-level support to deal with the added `exploded` keyword. Other reasonable approaches may or may not require a similar degree of compiler modifications.

The simplest choice is to provide no compiler support whatsoever, where developers are required to implement their own version of exploded objects using whatever techniques they desire. A slight improvement is to provide a design document that contains a description at some level of detail. For either of these choices, there is effectively no support for any particular set of features, such as allowing for inheritance or subsumption between Java and exploded object types, or between different exploded object types; there may not even be a notion of "types" for exploded objects after certain forms of implementations, such as when the transformation does little more than replace objects with groups of arrays and references with `int` indices into them.

Even if no compiler-level support is provided or required, it is still possible for an approach to provide suggestions or mechanisms to ease the conversion and implementation process. One such choice that would still not require any compiler-level support involves manually-translated approaches. Users could be provided with an implementation that does little more than perform the basics of the exploded object transformation by storing object fields in parallel arrays and replacing existing references with indices. Under this choice, users would at least have a framework from which to base further development, or to try to extend to support other features such as inheritance. It would probably not be practical to try to support subtyping between exploded and Java object types at the user-code level, for reasons described in Section 3.3 above, but it should be possible to at least support inheritance between different exploded object types through code and field copying (the "copy-down" strategy for inheritance). Supporting subsumption of exploded object types, where a subtype can be used where a supertype is expected, could also be implemented at the user-code level but would be somewhat cumbersome, likely involving explicit tracking of run-time type information in conjunction with index information. Adding support for garbage collection through user code

would be very difficult to do in an efficient fashion, and would be better reserved for choices that permit the modification of the virtual machine and preferably provide compiler support as well.

Adding a layer of library support on top of a manual approach can greatly simplify the programming interface for dealing with exploded objects. For example, library support might provide a Java `Object` wrapper for an exploded object instance, permit the use of convenient Java-style programming idioms such as iteration over collections, or otherwise conceal implementation details behind normal Java code, such as providing mechanisms for creating and deleting exploded object instances without requiring direct manipulation of the underlying arrays. However, library code would not be able to provide efficient support for garbage collection of exploded object instances, though it might aid attempts at implementing user-level garbage collection code by providing a cleaner interface as compared to manual approaches.

Choosing to modify or extend the language to add support for exploded objects can greatly facilitate a broad group of features. By adding keywords to the language to allow for tagging of classes as being targets of the exploded object transformation, and making corresponding modifications to the compiler, it is possible to provide for a partially- or wholly-automated transformation. Language extensions may be more complex, such as providing a domain-specific mini-language that can be used to specify object layout at a granularity beyond that initially specified by the exploded object transformation. For example, a domain-specific layout extension could provide for explicit indications of how internal arrays should be organized to improve cache performance, or specify divisions across the arrays to facilitate breaking them apart for distribution across multiple processors. Language extensions could allow for direct support of inheritance and subsumption involving exploded object types, by providing mechanisms for specifying typing relationships at the language level and performing translations within the compiler to produce appropriate code. While similar modifications could be made through user-level code written as needed by developers, providing language and compiler infrastructure to support them would make such modifications much more useful.

Even if no modifications are made to the language, it is possible to provide increased user support through compiler extensions. For example, if there are restrictions on typing relationships between exploded and non-exploded object types, a modified compiler can provide compile-time warnings or errors that may be much more useful than general type mismatch errors that a normal compiler may produce. If there are technically legal but undesirable or dangerous uses of exploded objects

or Java objects used to support them, such as Java objects created as wrappers over the underlying exploded object representation in a system where such wrapper objects are intended to be ephemeral, a modified compiler can help detect such semantically dangerous but syntactically legal uses at compile time instead of forcing developers to program using certain conventions. For example, in the Iterator-Based Wrapper approach to exploded objects, as described in Section 3.8 below, clients can maintain a persistent reference to the wrapper object returned by an iterator over a collection of exploded objects, but if they do so, the results of execution will almost certainly be erroneous as the contents of the wrapper object are updated as a side effect of iteration. Compile-time warnings or errors are almost certainly more useful to developers as compared to code that compiles and executes without run-time errors but produces erroneous results.

3.5 Run-Time Environment Concerns

Another dimension by which different approaches can be differentiated involves the run-time environment required by the approach. The most significant aspect of run-time support involves the degree of support for garbage collection, though general changes to the Java virtual machine and support for reflection involving exploded object types are also issues. Choices relating to garbage collection involve requiring explicit deallocation of exploded object instances versus providing for automatic garbage collection, and management of the free list and internal structure of the parallel arrays used to store exploded object instances.

In the Ideal approach, exploded objects would be handled exactly as non-exploded objects are, in that exploded objects would be allocated the same way that normal Java objects are through the use of `new`, and instances of exploded objects would be subject to automatic garbage collection when appropriate. Without modifying the virtual machine, however, this degree of support is not practical. Instead, a variety of alternatives may be explored, some of which should not require a customized JVM.

If a decision is made to require explicit deallocation of exploded object instances, then no particular choices are required in terms of user support. Wholly manual approaches or library-level support can provide direct access to the internal arrays to support removing instances, or convenience methods that encapsulate the deletion process. Language modifications can provide a special

operator, such as **delete**, or provide a normal method with a particular name such as `destroy`, to be used to remove exploded object instances.

If explicit deallocation is required, then another choice must be made regarding dangling references to deleted instances. One simple option is to specify that the effect of accessing a previously-destroyed exploded object instance is undefined. Another option is to reset some or all fields of a deleted instance to a known state, such as a special error value, to give user code the option of checking for the presence of a dangling reference before dereferencing it. Attempts to access a deleted instance could be trapped and throw an exception, but this approach either requires that no internal reference ever be re-used, or that efforts to detect access of a deleted instance may not always be successful. One way to ensure that every attempt to access a deleted instance is trapped is to add another layer of indirection before the parallel arrays, where a reference to an exploded object is a guaranteed-unique handle that is in turn mapped to an index into the parallel arrays. This approach requires additional memory as compared to the other choices for dealing with dangling references, but could be implemented entirely as user-level code. Alternately, internal allocation of slots in the parallel arrays could be performed in a fashion that attempts to minimize re-using formerly-used slots, so that dereferences of dangling pointers would probably be detectable, but detection could not be guaranteed.

If automatic garbage collection of exploded object instances is desired, an efficient implementation will almost certainly require a customized virtual machine. Otherwise, "smart pointers" [13] could be implemented at the user-code level to support garbage collection, provided a mechanism besides Java's **new** is present to allocate exploded object instances, or that compiler modifications were made to handle invocations of **new** involving exploded object types in an appropriate fashion. Due to the additional layers of overhead that would be imposed by such an approach implemented in user-level code, negative performance implications are probable. In addition, the presence of separate garbage- collection models for exploded versus non-exploded object types, such as a smart pointer-based approach for the former and a mark-and-sweep generational collector as may commonly be present for the latter, may make tuning the garbage collection behavior of an application even more difficult. Tuning a Java application's garbage-collection behavior is not always straightforward, and would be no easier in the presence of two different garbage-collecting schemes that may interact in non-obvious fashions.

A related decision involves management of the free list of slots in the internal parallel arrays that store exploded object instances, and management of the arrays themselves. As exploded object instances are deleted, either explicitly or through automatic garbage collection, "holes" (unused entries) will be created in the parallel arrays. To make efficient use of memory, these holes must be filled in with subsequent allocations, or the holes must be eliminated by reorganizing and compacting the contents of the arrays. If the choice is made to fill holes with subsequent allocations, a free list must be maintained to keep track of the unused locations. Alternately, an unused entry in the arrays could be filled with a special value, or an extra bit indicating used/unused state could be associated with each slot, and subsequent allocations could search through the parallel arrays to find the next available slot. Each choice may increase memory usage and have performance costs, with an explicit free list likely providing the smallest impact on performance but the largest memory cost, and storing special values directly in data elements requiring no additional memory but potentially large amounts of additional processor time needed to search through the arrays to find a slot with the special value. Each of these choices can be supported by user-level code, without requiring a customized virtual machine.

A different alternative would be to avoid the presence of holes entirely, by compacting or otherwise reorganizing the contents of the parallel arrays to ensure that all allocations are contiguous even after deletions. This choice would either require adding a level of indirection between exploded object references and the array indices to which they refer, such as having an exploded object reference be a handle which is then mapped to an array index through a lookup table, or would require the ability to identify all current indices and replace them with modified values after compacting the parallel arrays. The former approach would require additional memory for the handle to index lookup table, but could be implemented as user code. The latter approach may require virtual machine modifications to ensure that index values anywhere in the program can always be identified, and could require substantial processing to find and modify every index after compacting the parallel arrays.

A modified VM is required to provide for reflection involving exploded object types, as only the run-time environment within the JVM is allowed to create `Class`, `Method`, and `Field` instances from the `java.lang.reflect` package. Virtual machine modifications greatly facilitate supporting synchronization on individual exploded object instances, as the VM could, for example,

efficiently allocate locks for use with an instance only as required, versus the standard Java approach of providing one lock for every object instance; this is a variant of the lock nursery concept as described by Bacon, et al. [4]. Modifications to the virtual machine may also make it possible to efficiently support full interchanging of exploded and Java object references, including inheritance between them and subsumption between exploded and non-exploded types.

However, modifying the Java virtual machine should not be done lightly. Requiring a modified JVM negates one of the significant benefits of making use of Java, namely Java's portability. It may be much more difficult to benefit from future advances in virtual machine and just-in-time compiler technologies if a customized VM is required, as any advances provided by others must be integrated into the customized VM. While modifying the virtual machine may allow for a range of other choices that may not otherwise have been practical or possible, care must be taken to balance the negative effects of requiring a customized VM with any potential benefits.

In addition to support for garbage collection and general VM changes, support for reflection involving exploded object types is another aspect of run-time requirements of an approach. If decisions regarding the semantics of exploded objects dictate that reflection work in an identical fashion for both exploded and non-exploded types, then a customized virtual machine is required as user-level code cannot create instances of the classes found in `java.lang.reflect` that are returned by the normal reflection mechanisms. A slight change to semantics, and possibly syntax, would be to provide similar yet independent mechanics for reflection involving exploded object types; such reflective behavior could involve only user-level code.

3.6 Environment Requirements and Concerns

The fifth dimension along which approaches to the exploded object translation problem can be differentiated involves general environment requirements, such as development and debugging of programs making use of exploded objects. Environment-related options include providing no additional support, providing simple scripts or tools to assist with different aspects of the development process, or providing fully-integrated IDE support for application development, execution, testing, and debugging.

Under the Ideal approach, there are no special environment requirements or concerns, as any

existing environments and tools can be used directly, providing they can work with the customized VM that the Ideal approach would require. As one begins to move away from the Ideal approach through the compromises and modifications as described in previous sections, however, environmental requirements and concerns begin to arise.

Providing no additional support is the simplest of the options, for obvious reasons. An incremental improvement would be to provide scripts, tools, or other harnesses that help automate different aspects of the development process. One example would be to provide a wrapper that automates the compilation process if the compilation process takes more than one step, or scripts that automatically modify file locations or their contents to make normal compilers or debuggers easier to use.

A more involved option would be to provide a multi-stage translation process that generates versions of the code that are more debugger-friendly than the final version which may not be intended to be human-readable. For example, if the semantics of exploded objects or collections of exploded object instances are slightly or completely different than those of normal Java objects, the translation process may be able to produce intermediate forms of the exploded objects or collections of them that obey the different semantics but are otherwise normal Java objects. The intermediate version could be used with normal Java debuggers to help ensure that an application obeys the altered semantics of exploded objects in a given approach. A related option would be for the translation process to produce Java bytecode that includes debugging symbols and other information to try to facilitate the use of normal Java debuggers.

The most comprehensive option with respect to the environment would be to provide fully-integrated support for exploded objects within an existing IDE, such as Eclipse [34]. In such an approach, all issues relating to compiling, translating, and debugging an application that uses exploded objects would be as integrated with the development environment as are the equivalent tools for normal Java code.

To help define and round out the problem space as framed by the above five categories, three additional approaches to the exploded object transformation are presented below. Each approach is placed within the problem space, and then potential modifications to the approach are discussed to evaluate the regions of the problem space surrounding the approach.

3.7 The Ad Hoc Approach

At one end of the spectrum of compiler support options is the "Ad Hoc" approach to the exploded object transformation process. The Ad Hoc approach is an entirely manual approach that involves explicit, manual conversion of object types into untyped collections of parallel arrays containing their fields, with no effort being made to support any typing relationships between Java and exploded object types, or between different exploded object types–in fact, after the conversion process, there are no longer any "types" of exploded objects, just collections of fields and raw `int` indices into them. This approach requires explicit deallocation of individual exploded object instances, but as the entire process occurs at user-level code, no library, compiler, or JVM modifications are required or provided. Syntactically, the Ad Hoc approach to exploded objects is fundamentally different than normal Java objects, as user-level code directly manipulates arrays and indices instead of objects. There are no requirements for run-time support or environmental support for the Ad Hoc approach, and no such support is provided. The Ad Hoc approach is what was first used with UrbanSim to obtain the original performance benefits from exploded objects as described in Section 2.3, on page 7.

As substantial modifications would be required to preserve the notion of exploded object types under the Ad Hoc approach, including compiler or JVM modifications, providing different typing issues is not really a practical modification. Instead, the most practical way to modify the Ad Hoc approach would be to provide at least some degree of insulation from the raw parallel array representation. One such modification would be to add library code that presents an interface that permits some use of the normal Java collection and iteration idioms, as is found in the Iterator-Based Wrapper Approach, described below.

3.8 The Iterator-Based Wrapper Approach

More recent work on UrbanSim has resulted in the creation of a set of libraries that overlay the Ad Hoc approach used previously and provide a more convenient interface using iterator-based wrappers. Library code allows for clients to obtain a collection of exploded object references and then an iterator over that collection. The iterator yields what looks like a normal Java object, in that it has methods that can be invoked, fields that can be accessed, etc., and it serves as a wrapper to the

underlying exploded object instance.

While the presence of libraries increases the degree of support as compared to the Ad Hoc approach, there is still no compiler-level support. In particular, users must be cautious about how they make use of the wrapper objects. As the iterator is advanced to step through the collection, the *same* Java object is returned each time, but its internal index referring to the parallel arrays is modified. This approach requires only a single Java `Object` be created to handle iterating over a large number of exploded object instances, which is important; creating one Java wrapper object per exploded object instance would at worst result in the presence of one Java `Object` instance for each exploded object instance, eliminating the benefits of the exploded object transformation process, or would result in substantial object "churn" as a different wrapper object would be created and subsequently destroyed with each step through the collection. As a result of the iterator returning the same Java `Object` with each call, if user code maintains a persistent reference to the wrapper object, subsequent iteration through the collection will cause the wrapper object's contents to be updated as a side effect. That behavior is almost certainly not what the user code would be expecting, and is markedly different from the normal semantics of iteration over collections in Java.

The Iterator-Based Wrapper approach is implemented entirely as library code sitting on top of the Ad Hoc approach. As such, it cannot provide any support for detecting dangerous uses of the wrapper objects, such as maintaining a persistent reference to one. The approach is also unable to introduce any significant changes to typing issues as compared to the Ad Hoc approach; it is unable to support inheritance or subsumption between exploded and non-exploded object types, or between different exploded object types. The Iterator-Based Wrapper approach can, however, provide limited typechecking to ensure that references obtained from a collection are not directly compared with or cast to other exploded object types, as the type of the wrapper object is distinct for each exploded object type being wrapped. For example, the wrapper object type for a `Household` exploded object type is distinct from the wrapper object type used for a `GridCell` exploded object type. In the Ad Hoc approach, both references would be raw **`int`** values indexing into the parallel arrays, and as such would be of identical types as far as normal Java typechecking is concerned.

Much like the Ad Hoc approach, significant modifications would have to be made to the Iterator-Based Wrapper approach to allow it to support different typing options or garbage collection, likely including compiler and/or virtual machine changes. However, one very useful and relatively

straightforward modification to the Iterator-Based Wrapper approach that could be made if one is willing to modify the compiler is to add checking for dangerous uses of the wrapper object returned by iterators over collections of exploded objects. In particular, compiler analyses could determine if any persistent references are made to the wrapper object, and generate compiler errors or warnings instead of requiring programmers to spot the erroneous results from improper use of the wrapper object under the basic Iterator-Based Wrapper approach.

It is due to these concerns that arise from a decision to not modify the JVM that the Restriction Approach has been created.

3.9 The Restriction Approach

The Restriction Approach is an approach to the exploded object transformation process that has three key requirements:

1. Must run on a standard Java virtual machine

2. Must provide significant performance improvements over the non-exploded approach

3. Should be easier and less error-prone to use than the Ad Hoc and Iterator-Based Wrapper approaches

Of these requirements, the first is the most constraining. As described previously, it is not practical to provide for full subsumption between exploded and non-exploded types, or to provide garbage collection, without modifying the virtual machine. Likewise, reflection on exploded object types is impossible, as is efficient support for synchronization at the instance level. The second requirement is addressed by the underlying exploded object representation, in that the exploded object version of an application should use less memory as compared to the original Java `Object` version, and performance gains should arise from decreased memory requirements and specific choices made during the code translation process. The third requirement implies that a higher degree of support should be provided as compared to either the Ad Hoc or Iterator-Based Wrapper approach, so the Restriction Approach involves modifications to the compiler and language, as well as providing some environmental support to assist with the translation and application debugging processes.

As a result of the requirement to not modify the Java virtual machine, the Restriction Approach imposes four restrictions on the usage of exploded objects:

1. Explicit deallocation of exploded object instances is required.

2. There is no support for per-instance synchronization involving exploded object types.

3. There is no support for reflection involving exploded object types.

4. Java `Object` references cannot be assigned to, or explicitly or implicitly cast to, exploded object references, and vice versa.

Under these restrictions, which give the Restriction Approach its name, the approach provides automated source-to-source translation, and supports nearly all Java `Object` semantics for exploded objects, such as fields of Java or exploded object types, methods which can be invoked, passing of exploded object references to methods or returning them from methods, and efficient support for collections of exploded objects and iteration over those collections.

In terms of implementation concerns relating to compiler support, the Restriction Approach makes use of a modified compiler to perform analyses that ensure the restrictions of the approach are obeyed as well as to automatically generate code. It also provides a range of library-level support through specialized types used to efficiently store collections of exploded object references. It requires a small syntactic modification to the language to allow for tagging of exploded object types by adding the **`exploded`** keyword to the class definition.

With respect to semantics and typing concerns, exploded object types are wholly separated from non-exploded types; no inheritance or subsumption is permitted between the two kinds of types. However, support for both inheritance and subsumption between different kinds of exploded types is permitted, by automatically tracking run-time type information and referencing it as needed to perform dynamic dispatching. No interfaces can be implemented by exploded object types.

As in the Ad Hoc and Iterator-Based Wrapper approach, garbage collection is not used with exploded object instances; instead, explicit deallocation is required. Reflection involving exploded object types is not allowed, and the JVM is not modified in any way.

The Restriction Approach is intended to provide considerable flexibility and preservation of Java `Object` semantics for exploded object types to the extent possible without requiring a customized virtual machine. Modifying the approach to add support for garbage collection, or to allow subsumption between exploded and non-exploded types, would require a customized virtual machine for reasons described in Section 3.3. One modification that could be made without requiring the addition of a customized virtual machine would be to allow exploded object types to implement distinct interfaces, such as specially-tagged exploded interfaces that cannot be implemented by non-exploded types. Exploded objects could be cast to interface types they implement by tracking run-time type information with each reference to an exploded interface type. This modification would bring the semantics of exploded objects closer to the semantics of normal Java objects by allowing interface implementation.

Chapter 4 describes the Restriction Approach, its goals, its features, and its restrictions in more detail. Subsequent chapters present the semantics of the Restriction Approach as well as an implementation strategy that has been used to create a prototype implementation for purposes of evaluating the Restriction Approach.

Chapter 4

THE RESTRICTION APPROACH

This chapter presents an overview of the Restriction Approach by describing its major goals, the restrictions it imposes, and the general features it provides, expanding on the description presented at the end of the previous chapter.

4.1 Goals of the Restriction Approach

The Restriction Approach is intended to be a realization of the exploded object transformation process that meets the following goals:

1. Must run on a standard Java virtual machine

2. Must provide significant performance improvements over the non-exploded approach

3. Should be easier and less error-prone to use than the Ad Hoc and Iterator-Based Wrapper approaches

Any language extensions, new language features, and even new libraries face stiff opposition to their adoption in that they must replace whatever tools are in current use. For the purposes of this work, requiring a customized Java virtual machine, or other customized lower-level support such as a native-mode compiler, is simply too much to expect of a typical user of the system. Tying the approach to a specific JVM or platform(s) for which a customized native-mode compiler were made available would drastically curtail the approach's portability and general applicability.

The UrbanSim urban land use and transportation modeling system presented in Chapter 2 serves as a concrete, real-world application that has demonstrated the potential benefits from the exploded object transformation process. Any new approach to the exploded object transformation process should provide generally comparable performance benefits for similar computational processes.

The exploded-object approach as described in this work should result in a run-time performance benefit of at least a factor of two over a traditional Java `Object`-based approach, and should reduce memory usage by at least one-half provided the objects in the domain are generally small and/or contain small fields such as **`byte`** or **`short`** fields.[1]

The third goal of this work is that it provide a system that does not place a significant burden on developers making use of it, as compared to the Ad Hoc and Iterator-Based Wrapper approaches to exploded objects as described previously. In particular, this means that certain approaches that add considerable complexity to the specification and/or implementation of exploded objects are unsuitable candidates. This criteria also imposes a minimum degree to which exploded objects should preserve normal Java `Object` semantics; if the current or previous approaches to exploded objects as found in UrbanSim do a better job of preserving Java `Object` semantics, then this work will have fallen short.

4.2 Restrictions of the Restriction Approach

As discussed in Chapter 3, implementation concerns affect many choices that determine what features can be provided by an approach and what tradeoffs are practical. By requiring no modifications be made to the Java virtual machine, the Restriction Approach accepts certain limitations on what it can provide in an efficient fashion. The four restrictions on the use of exploded objects imposed by the Restriction Approach are listed below, along with explanations of why they are imposed.

1. Explicit deallocation of exploded object instances is required.

2. There is no support for per-instance synchronization involving exploded object types.

3. There is no support for reflection involving exploded object types.

4. Java `Object` references cannot be assigned to, or explicitly or implicitly cast to, exploded object references, and vice versa.

---

[1]These goals may seem weak, given the potential gains presented in Section 2.4, but it is believed that they are reasonable expectations for a non-contrived example in a highly complex target application, continuing `Object`-specific advances in VM implementations and just-in-time compiler technology, etc.

Supporting automatic garbage collection of exploded object instances through user-level code is likely to be impractical at best. In addition, explicit deallocation of object instances is a common pattern of one category of domains most likely to benefit from the exploded object transformation process, namely simulation domains such as UrbanSim and others as described in Chapter 2. Concerns of data consistency often require explicit deallocation or special handling of a simulation entity object prior to its removal from the system. For example, explicit relationships between different simulation entities, such as between a household and the house in which it resides, must be updated before a simulation entity object is removed. Aggregation or summaries of simulation entity characteristics, such as population totals based on household counts and sizes, must be updated as well prior to removal of an entity from the system, or the aggregated data will become inconsistent. This domain property, combined with the implementation difficulties in supporting automatic garbage collection without modifying the virtual machine, results in the decision to require explicit deallocation of exploded object instances.

Supporting synchronization on individual exploded object instances is another feature that is difficult to provide in an efficient fashion without modifying the virtual machine. Likewise, allowing reflection on exploded object types cannot be performed without virtual machine modifications, as discussed in the previous chapter. In typical simulation applications, or other environments in which very large numbers of particular types of objects are present, it is extremely uncommon to wish to use either per-instance level synchronization or reflection. These restrictions are therefore consistent with typical domain characteristics, and follow naturally from the requirement of an unmodified JVM.

The lack of support for inheritance or subsumption between exploded and non-exploded object types is also derived from the requirement that a standard virtual machine be used. While the distinction between exploded and non-exploded types may impose some additional burden on developers, the Restriction Approach does allow for both inheritance and subsumption between different exploded object types, so those aspects of object-oriented programming are preserved for exploded object types.

4.3 Features of the Restriction Approach

Operating under the restrictions listed previously, the Restriction Approach provides a range of features and benefits as compared with the Ad Hoc and Iterator-Based Wrapper approaches described in Sections 3.7 and 3.8, respectively. The major features provided by the Restriction Approach are:

1. Support for most Java `Object` semantics for exploded objects
2. Compiler-level support to identify problems
3. Automation of the transformation process
4. Support for the normal Java idiom of collections and iteration over collections of exploded objects

Apart from the restrictions imposed by the Restriction Approach, the approach supports normal Java `Object` semantics for exploded objects. Exploded objects look and act like Java `Objects` in nearly every way, including fields, methods, and inheritance and subsumption. Exploded objects have fields just like Java `Object` subclasses, including fields of primitive types, Java `Object` types, other exploded object types, and arrays of those types. Exploded objects have methods as well, including static and instance methods, methods can be invoked on instances of exploded objects, exploded objects can be passed as parameters to methods found within exploded and non-exploded types, and exploded object references can be returned from methods as well. Inheritance between different exploded object types is supported, as is subsumption between exploded object types so that a subclass can be used when a superclass is expected. Dispatching of method invocations based on the run-time type of an exploded object reference occurs just as for normal Java `Object` references. Chapter 5 contains more details concerning the semantics of the Restriction Approach.

Compiler modifications allow for the introduction of analyses that verify that the restrictions of the Restriction Approach are being obeyed, as well as automating the vast majority of the transformation process. Chapter 6 contains more information concerning the implementation details of the prototype implementation of the Restriction Approach.

The Restriction Approach includes support for specialized collections of exploded objects, allowing efficient support for the normal Java idioms of collections and iteration over collections. Unlike the Iterator-Based Wrapper approach described in Section 3.8, there are no restrictions on how the object returned by the iterator can or must be used as the compiler infrastructure provides for appropriate translations. Chapter 7 describes the template-based support for specialized collections in more detail.

Chapter 5

SEMANTICS OF THE RESTRICTION APPROACH

This chapter continues the description of the Restriction Approach as presented at the end of the previous chapter, concentrating on the semantics of the approach. The Restriction Approach consists of three aspects: specification of exploded objects, semantics of exploded objects including how they may be used, and an implementation specification as to how this approach is realized. This chapter discusses the first two of these aspects, along with a justification of the semantics for exploded objects, and includes several examples of translation and usage to help ground the discussion for this and future chapters. An implementation strategy for the Restriction Approach is presented in Chapter 6.

Together, the restrictions on exploded object usage allow for statically unambiguous object layout based on static type information. This information is exploited entirely within the compiler, providing performance benefits while hiding most details from the programmer, as described below.

5.1 Specifying Exploded Objects

In order to minimize the impact of working with exploded objects upon developers, exploded objects are specified on a per-class basis; that is, all instances of a particular class will either be exploded or will be normal Java objects. This avoids confusion that may arise from mixing exploded- and non-exploded instances of a single class. An exploded object `Household` class is defined using the new **`exploded`** keyword, as illustrated in Figure 5.1.

5.2 Translation and Usage Examples

Consider a `Household` object used to store the size, age, and income category of a household. To designate the type as an exploded type, the **`exploded`** keyword is added to the definition, as shown in Figure 5.1. All of the rest of the definition is unchanged, remaining normal Java code. This causes

a canonical collection to be generated during the translation process. The canonical set stores all instances of the `Household` type as entries in a parallel array, so that a reference to a `Household` becomes an index of type **`int`** into the parallel arrays. Figures 5.2 and 5.3 contain a partial definition of the canonical set used to store `Household` instances. See Section 6.1.2 for more details on the canonical sets used to store instances of an exploded object.

Accessing a field in an instance of an exploded object is translated into an access into the corresponding array in the canonical set, and an instance method invocation is translated into a method invocation involving the canonical set, as shown in Figure 5.4. Note that the original `Household` reference `hh` has been converted into an **`int`** `hh_idx` used to index the array, and that creation of an instance of an exploded type involves invoking the `create` method from its canonical set instead of using Java's **`new`** operator.

Specialized collections used to hold references to exploded object instances are provided as part of the translation process. Selection of the appropriate specialized collection is currently a manual process in the prototype implementation. Specialized collections provide interfaces analogous to those found in Java collections such as `java.util.Set`, including iteration. Figure 5.5 illustrates the use of a specialized `Set` to store `Household` references and how that usage would be translated. The middle portion of the figure shows the user-level usage of a specialized `Set`, and the bottom portion shows what results from the full translation of the specialized types into forms that preserve the original type in name only (e.g., `Set_Household`) and store only **`int`**s. Section 6.1.1 contains details on specialized parameterized collections and their usage.

5.3 Semantics of Exploded Objects

This section describes the semantics of exploded objects as defined by the Restriction Approach.

To add exploded objects to Java, this approach adds a third fundamental type to the current Java type system. All data types are either Java objects (reference types), values of primitive types, or exploded objects. All exploded objects can be thought of as inheriting from an `Exploded-Object` base type, which, much like Java's `Object` base type, defines methods common to all exploded objects. In practice, there is not an actual `ExplodedObject` type required by the exploded object transformation process in general, or provided by the Restriction Approach in particular.

```
1 public exploded class Household {
2   private int ID;
3   private byte size;
4   private byte age;
5   private byte incCat;
6
7   private static int NEXT_HH_ID = 0;
8
9   public static int getNextID() { return NEXT_HH_ID; }
10
11  public Household() {
12    ID = Household.NEXT_HH_ID++;
13    size = (byte) 0;
14    age = (byte) 0;
15    incCat = (byte) 0;
16  }
17
18  public int getID() { return ID; }
19  public byte getSize() { return size; }
20  public byte getAge() { return age; }
21  public byte getIncomeCategory() { return incCat; }
22  public void setSize(byte size) { this.size = size; }
23  public void setAge(byte age) { this.age = age; }
24  public void setIncomeCategory(byte ic) { incCat = ic; }
25  public String toString() { return new String("HH#" + getID()); }
26 }
```

Figure 5.1: Definition of an exploded `Household` object. Note that the definition is normal Java code in every way, apart from the introduction of the **`exploded`** keyword.

```
1 public class CanonicalSet_Household implements FilterSet<Household> {
2   // Fields from original Household object
3   private int[] ID;
4   private byte[] size, age, incomeCategory;
5   public static int NEXT_HH_ID = 0;
6   public static int getNextID() { return NEXT_HH_ID; }
7
8   // Static reference to the canonical set
9   public static CanonicalSet_Household ref;
10
11  // Method to initialize the canonical set
12  public static void init(int initSz) {
13    ref = new CanonicalSet_Household(initSz);
14    ...
15  }
16  // Instance methods from original Household object.
17  public int getID(int idx) { return ID[idx]; }
18  public byte getSize(int idx) { return size[idx]; }
19  ...
20  public void setSize(int idx, byte size) { size[idx] = size; }
21  ...
22  // Methods for Set<Household> interface
23  // Note that the specialized types have already been converted into
24  // their final form involving raw indices, not the Household-level
25  // versions exposed to programmers.
26  public Iterator<int> iterator() { ... }
27  public boolean isEmpty() { ... }
28  public boolean contains(int idx) { ... }
29  public int size() { ... }
```

Figure 5.2: First part of the implementation of the canonical set in which exploded `Household` objects are found. Note that the canonical set as shown above is generated by the translation process; the version as exposed to programmers contains specialized types such as `Iterator<Household>` and `contains(Household)` instead of `Iterator<int>` and `contains(int)`.

```
30  // Method for FilterSet<Household> interface
31  public FilterSet<int> filter(Filter<int> filt)
32    { ... }
33
34  // Methods for creation/deletion of instances
35  // Note that one create(...) method is supplied for each constructor
36  // in the original exploded object definition, with field
37  // assignments the same as found in the constructor.
38  public int create() {
39    int nextIdx = getNextFreeIdx();
40    ID[nextIdx] = NEXT_HH_ID++;
41    size[nextIdx] = (byte) 0;
42    age[nextIdx] = (byte) 0;
43    incomeCategory[nextIdx] = (byte) 0;
44    return nextIdx;
45  }
46
47  public void destroy(int idx) { markIdxAsFree(idx); }
48
49  // Methods for managing indices for allocation/deallocation purposes
50  private int getNextFreeIdx() { ... }
51  private void markIdxAsFree(int idx) { ... }
52
53 }
```

Figure 5.3: Second part of the implementation of the canonical set in which exploded `Household` objects are found.

```
1 // Original Code
2 Household hh = new Household();
3 hh.setAge((byte) 3);
4 int hhID = hh.getID();
5
6 // Translation
7 CanonicalSet_Household hhCSet = CanonicalSet_Household.ref;
8 int hh_idx = hhCSet.create();
9 hhCSet.setAge(hh_idx, (byte) 3);
10 int hhID = hhCSet.getID(hh_idx);
```

Figure 5.4: Accessing instance fields and methods of an exploded object. It is assumed that the canonical set has a static reference `ref` through which all accesses occur.

Requiring an actual `ExplodedObject` type as part of exploded objects would require support for inheritance and subsumption involving exploded object types, which may not always be desirable due to implementation concerns described in Chapter 3. Under the Restriction Approach, user code uses specialized collections of specific types of exploded objects, and casting to `ExplodedObject` is never needed. This approach provides for the presence of more precise type information and the elimination of a layer of inheritance which would otherwise require run-time type information to be associated with many more references than may actually be required.

Exploded objects contain fields and methods just like Java objects, with the same allowable access control keywords of **`public`**, **`private`**, and **`protected`**. Fields and/or methods can also be declared to be **`static`** and/or **`final`** with the same semantics as those for **`static`** or **`final`** fields or methods in Java objects. There are no limitations on allowable types of fields in an exploded object, or of parameter types or return types for its methods, or of uses of exploded object types as fields or parameters involving non-exploded object types, save the general restrictions on mixing exploded- and non-exploded object references as described below.

Exploded object types cannot implement interfaces, unlike Java `Object` subclasses. Exploded object classes may inherit from other exploded object classes, but not from any non-exploded object classes. Non-exploded object classes cannot inherit from exploded ones. All exploded object classes

```
1 // Original code
2 Set allHHSet = ...
3 Set highIncomeSubset = new HashSet();
4 Iterator iter = allHHSet.iterator();
5 while ( iter.hasNext() ) {
6  Household hh = (Household) iter.next();
7  if ( hh.getIncCat() > INCCAT_MEDIUM )
8    highIncomeSubset.add(hh);
9 }
10 // Intermediate version (converted by programmer)
11 Set<Household> allHHSet = CanonicalSet_Household.ref;
12 Set<Household> highIncomeSubset = new HashSet<Household>();
13 Iterator<Household> iter = allHHSet.iterator();
14 while ( iter.hasNext() ) {
15  Household hh = iter.next();
16  if ( hh.getIncCat() > INCCAT_MEDIUM )
17    highIncomeSubset.add(hh);
18 }
19 // Final translation of intermediate version (from compiler)
20 Set_Household allHHSet = CanonicalSet_Household.ref;
21 Set_Household highIncomeSubset = new HashSet_Household();
22 Iterator_Household iter = allHHSet.iterator();
23 while ( iter.hasNext() ) {
24  int hh_idx = iter.next();
25  if ( hhCSRef.getIncCat(hh_idx) > INCCAT_MEDIUM )
26    highIncomeSubset.add(hh_idx);
27 }
```

Figure 5.5: Usage of specialized `Set` collection to store `Household` references. The first part of the figure shows the original Java version. The middle portion shows the intermediate version after the programmer has modified it to make use of specialized collections. The bottom portion shows the final version that is produced by the compiler, including instantiation of the specialized collection templates which will store **ints**. It is assumed that `hhCSRef` is a reference to the canonical set containing all `Household` instances.

conceptually extend the `ExplodedObject` base class, as described in Section 5.3.1, on page 48.

Instances of exploded objects exist only within parameterized collections known as canonical sets. One important aspect of the Restriction Approach is that it requires parameterized types that support heterogeneous translation to handle efficient storage of Java primitive types; such storage is used extensively by generated code to provide equivalents to the common Java idioms of collections and iteration involving exploded object references. Parameterized types such as those found in GJ [5] are not sufficient, as they do not support heterogeneous translation, but those found in Centralia [31] are sufficient.

The canonical sets contain methods to create and destroy instances, and to return parameterized collections of references to subsets of the instances stored in the collection. A `create` method is provided to allocate new instances of exploded objects, with the **`new`** operator used to create arrays of references to exploded object types. The Restriction Approach will automatically transform the use of **`new`** to allocate an exploded object instance into a call to the appropriate `create` method, but programmers are encouraged to make direct use of `create` to help them identify locations where specialized collections may also be of use. Exploded objects as described in this work are not intended to be suitable for frequent creation and deletion, but rather in cases where they are long-lived, as is common in simulation domains. See Section 5.3.2 for more information concerning these parameterized collections.

Like Java `Objects` and primitive types, references to exploded objects cannot be mixed with references to non-exploded objects; they cannot be assigned to each other, compared, or cast to a class type that extends `Object`. The special reference value of **`null`** has the same meaning for an exploded object reference as it does for a reference to a normal Java object. A reference to an exploded object can be cast to another exploded object type, provided the cast is legal given the inheritance relation between the objects. See Section 5.3.3, on page 50 for more details.

Exploded object references may be used as parameters to methods, and/or as return types, with no restrictions other than the general restrictions governing mixing exploded- and non-exploded object references and casts.

Only limited run-time type information is provided for exploded object references. Dynamic dispatching works for exploded object references as for Java objects, as does **`instanceof`** (subject to the normal restrictions about mixing exploded and non-exploded objects), but no other forms of

reflection are allowed with a reference to an exploded object.

Synchronization using an exploded object as a semaphore is not allowed; there are no equivalents of Java `Object`'s `wait` or `notify` methods for exploded objects. Collections of exploded object references do not have this limitation, provided that the collections themselves are not exploded object types.[1]

There is no support for mixing legacy code using non-exploded objects of the same type (name) as exploded objects, as they are of different types. All source code involving exploded objects must be available to the compiler at compile-time, so that all references to exploded objects can be translated. This does not preclude the use of libraries or precompiled classes that do not make use of exploded objects in any way, however.

5.3.1 Methods and Fields of Exploded Object Base Type

All exploded object types conceptually extend the `ExplodedObject` base class. The `ExplodedObject` base class defines methods common to all exploded object types in the same way that the `Object` base class defines methods common to all Java `Object` subclasses. The `ExplodedObject` base class contains two methods: **`public`** `String toString()`, which returns a `String` object containing a representation of this object, and **`public int`** `hashCode()`, which returns a hash code value for this object. Either of these methods can be overridden by any specific exploded object type.

5.3.2 Parameterized Collections for Exploded Objects

All instances of exploded objects exist solely within a parameterized collection. This allows all instances to be represented in a single set of parallel arrays with minimum overhead while providing a convenient interface for accessing them in fashions typical of UrbanSim-style simulations such as iteration or subset extraction. One canonical collection exists for each exploded object type, and it is an unordered set that supports filtering based on a **`boolean`** predicate, as in `FilterSet<T>` for a collection of exploded object type `T`. These collections provide methods for iterating over their

[1] It is unlikely that collections would be converted to exploded object types in general. There may be some highly specialized cases where such a conversion may be beneficial, such as when very large numbers of very small sets are present.

elements, retrieving sub-collections, creating new instances, destroying existing instances, etc. See Figures 6.1 and 6.2 for details on the `FilterSet<T>` interface implemented by the collection.

In addition to the methods defined in `FilterSet<T>`, the canonical collection also defines methods for creating and destroying instances of exploded objects, and for comparing one instance of type `T` with another. Note that these methods are described at the programmer level, to illustrate what interface is exposed to developers, and not at the underlying translation level.

`public T` `create(...)` One `create(...)` method exists for each constructor definition in type `T`, with corresponding parameters. Invoking `create(...)` causes a new instance of `T` to be created inside the collection, effectively invoking `T`'s constructor with the same signature as the `create(...)` call. The reference to the exploded object, in the form of an index that can be used with the canonical set, is returned by the `create(...)` call. Developers are encouraged to use the `create(...)` method to create new instances of exploded object type instead of Java's **`new`** operator, as using `create(...)` provides a reminder that there must be a corresponding `destroy(...)` invocation (described below), and encourages the use of specialized collections to store the reference to the instance. The prototype implementation automatically replaces invocations of the **`new`** operator involving exploded object types with a call to the corresponding `create(...)` method, however.

`public void` `destroy(T oe)` Removes the instance of `T` referenced by `oe`. Any subsequent attempts to access or dereference `oe` will result in undefined behavior. Implementation options for the effects of accessing a destroyed instance include accessing an instance with possibly-invalid data, accessing an instance with always-invalid data (such as initial values, object-specific error values, or all zeros), or an exception being thrown. If an exception is to be thrown, an explicit check on every access must be added by the implementation, which can be quite expensive. In practice, accessing a destroyed object instance should only happen in the same cases as when the original untranslated application would have generated a `NullPointerException` such as accessing an element of an array which has been manually removed by setting the index to **`null`**, or when the original application would have exhibited undesirable or erroneous behavior by accessing an instance which should no longer be present. Neither of these cases should happen in a final-stage application. Accessing a destroyed object in the prototype implementation results in possibly-invalid data being accessed.

public boolean `equals(T oe1, T oe2)` Compares the two exploded object instances (using the single-argument `equals` method defined for the exploded object type, if one is provided, or a default comparison test based on internal instance index if not). The `equals` method is placed into the parameterized collection for the exploded object type, instead of remaining as a single-argument method of the original type, to allow for additional static type-checking to avoid unnecessary casts as are often found in implementations of `equals(Object)` in normal Java objects.

5.3.3 Interfaces and Casts

While exploded objects can implement interfaces, cannot be cast up to the interface type. The only effect of an exploded object type implementing an interface is to ensure that the exploded object type implements all the methods defined in the interface.

Exploded object references cannot be cast to non-exploded types, including interfaces they implement; a compile-time error is reported. Casts between exploded types are handled by the insertion of a run-time check, if necessary, or generate a compile-time error if they cannot possibly succeed. A cast that is illegal at run-time throws a `ClassCastException` just as is thrown by an illegal run-time cast of a Java object.

5.4 Justification of Proposed Semantics for Exploded Objects

When presenting the justification for the proposed semantics for exploded objects, it is necessary to view them in the context of the expected use of exploded objects.

It is expected that exploded objects, as presented in this work, will be applied after an application has reached a stable state to try to improve the performance and/or decrease the memory usage of an existing application. In such cases, it is safe to assume that all application source code will be available, including object definitions, though some well-encapsulated libraries may be available only in bytecode form. Ideally, developers would profile their application to determine which object type(s) may be good candidates for conversion to exploded objects and then make the necessary changes, which should tend to be limited in scope. The Restriction Approach is not intended to be suitable for use with incremental development of an application.

5.4.1 Tripartite Type Hierarchy

The decision to have exploded objects exist in a separate type hierarchy from Java objects is based on a desire to reinforce to users that exploded objects are fundamentally different from normal Java objects. Semantics of exploded objects, while mostly similar to that of normal Java objects, differ in several important fashions, and it seems better to reflect these differences through the type system at a low level rather than through run-time checks. Additionally, treating exploded object types as a separate kind of type allows for unambiguous object layout based on static type information that is automatically exploited by the compiler. If exploded objects were treated as a special kind of normal Java objects, it would not always be possible to determine what object layout was used for a particular reference, at least probably not to a degree that would allow for adequate performance without requiring modification to the Java virtual machine as described in Section 3.3

The inability to compare or mix references between Java objects and exploded objects should not impose an undue burden on programmers, given the typical usage patterns of simulation entity objects common in UrbanSim-style simulations. Inheritance hierarchies of simulation entity types are generally flat, with little if any use of dynamic type information, with that use restricted to dynamic dispatching.

5.4.2 Existence in Parameterized Collections

The basic idea behind exploded objects is to alter object layout by storing object fields in parallel arrays. Use of a different storage mechanism provides opportunities for reductions in memory usage as well as allowing for general performance improvements. Requiring all instances to exist solely in one location facilitates this style of transformation, and making the single collection a parameterized interface is useful in the application domains of interest.

Exploded objects, as described in this work, are intended for use in situations where objects are infrequently created and destroyed, and generally are very long-lived. This usage pattern is typical in UrbanSim, where all or nearly all simulation entities of a given type are created at the start of the simulation, or execution of a coarse-grained model component, and generally persist through the entire simulation. Simulation entities tend to be manipulated in a consistent fashion across model components, as well; typically, a collection containing a subset of the entities is obtained,

the subset is iterated over, and localized processing is performed on each entity. These patterns of infrequent allocation and deallocation and highly structured accesses lend themselves toward a storage mechanism separate from Java objects, which often vary enormously in terms of lifespan and access patterns even among objects of the same type across the scope of an application.

By mandating that all instances of exploded objects exist in a single, canonical location with the requirement of explicit deletion, it is much easier for programmers to ensure that no unnecessary instances are left in memory, as could happen if a reference to a collection containing Java object references were held longer than desired. Explicitly invoking `destroy` to destroy an exploded object instance guarantees that the instance is removed, whereas the only way to remove a normal Java object is to ensure that there are no references to it anywhere. In addition, locating all instances of an exploded object type in a single location ensures that a consistent, localized interface is provided for accessing subsets of elements, versus ad hoc approaches where simulation entity objects may "live" in many different data structures spread across many different pieces of code. This benefit could be obtained for normal Java objects through disciplined programming, but it is provided and enforced automatically as part of the Restriction Approach.

5.4.3 Handling of Interface Implementation and Casts

In the Restriction Approach, exploded objects cannot implement interfaces. Different choices of allowing interface implementation require varying levels of additional compile- and run-time support, with the largest additional support and increase in complexity being required for an approach which allows for implementation of an interface by both exploded and non-exploded types. To help justify the decisions made in the restriction approach and its accompanying prototype implementation regarding the implementation of interfaces by exploded types, it is useful to examine several possible alternatives in the context of an example.

For example, suppose there is an interface `PrettyPrintable`, as defined in Figure 5.6, and several classes that implement that interface, as shown in Figure 5.7.

Some code that uses these classes is shown in Figure 5.8. The code in Figure 5.8 is legal Java code, as both `PPBytePair` and `PPStringPair` implement the `PrettyPrintable` interface. The `PPBytePair` class is a prime candidate for replacing with an exploded object, as it

```
1 public interface PrettyPrintable {
2   public void prettyPrint();
3 }
```

Figure 5.6: Definition of the `PrettyPrintable` interface.

```
4 public class PPStringPair implements PrettyPrintable {
5   public void prettyPrint() { ... }
6   private String first, second;
7   public PPStringPair(String first, String second) {
8     this.first = first; this.second = second;
9   }
10  public String getFirst() { return first; }
11  public String getSecond() { return second; }
12 }
13
14 public class PPBytePair implements PrettyPrintable {
15  public void prettyPrint() { ... }
16  private byte first, second;
17  public PPStringPair(byte first, byte second) {
18    this.first = first; this.second = second;
19  }
20  public byte getFirst() { return first; }
21  public byte getSecond() { return second; }
22 }
```

Figure 5.7: Definition of two types implementing the `PrettyPrintable` interface.

```
23  public void debuggingPrint(PrettyPrintable pp) {
24    if ( PRINT_DEBUGGING == true )
25      pp.prettyPrint();
26  }
27
28  PPBytePair ppbp = new PPBytePair((byte) 1, (byte) 8);
29  PPStringPair ppsp = new PPStringPair("Alpha", "Beta");
30
31  debuggingPrint(ppbp);
32  debuggingPrint(ppsp);
```

Figure 5.8: Usage of mixed types implementing the `PrettyPrintable` interface.

contains two small (1-byte) fields, each of which will likely result in at least three wasted bytes per class instance on a typical JVM. Converting the `PPStringPair` class into an exploded version may not yield significant benefits, so for the sake of the example, suppose only the `PPBytePair` class and not the `PPStringPair` class will be converted into exploded object form. Converting just the `PPBytePair` class into an exploded class will result in the mixing of exploded and non-exploded types in the piece of code shown in Figure 5.8, through passing the two variables to the `debuggingPrint` method which takes a parameter of an interface type. There are several options regarding this mixing:

Disallow: Do not allow exploded objects that have been cast to interface types to be mixed with non-exploded objects that have been cast to interface types. In the example, a compiler error would be generated on the `debuggingPrint(ppbp)` invocation, as that call would result in the `debuggingPrint` method being invoked with both an exploded and a non-exploded type.

In this alternative, exploded and non-exploded object types can implement the same interfaces. Escape and alias analysis is required[2] to determine for all interface types appearing as the types of variables, parameters, and return values, whether the value may have been cast from an exploded

[2]Or, a more conservative approach would be to use static type information to determine whenever mixing *might* occur, such as through static analysis of **`implements`** declarations in exploded and non-exploded classes, or through global analysis of upcasts to interface types, and to explicitly disallow it under all circumstances.

object at any point in the past, and if so, to disallow the usage which causes mixing. State of the art analyses for Java such as those presented by Woo et al. [44] and Choi et al. [8] should be adequate to identify nearly all cases in practice. With this option, retrofitting an existing piece of code with exploded objects would potentially require an "all or nothing" approach, where all types interacting with a given variable or method be converted to exploded types, or none of them. This may result in the conversion of a great many types into exploded types, which may have undesired ramifications if the non-exploded types were used in one of the ways in which exploded types cannot be used, such as for synchronization purposes.

A related and somewhat cleaner option would be to disallow casting of an exploded object type to any interface it implements. With this option, analysis is greatly simplified, as static analysis of all upcasts to an interface type involving exploded objects would result in compile-time errors. This option may require substantial reworking of code that makes use of objects cast to interface types, however.

Independent: Require wholly independent interfaces be implemented by exploded and non-exploded objects, with no mixing of the interface types allowed. In the example, a compiler error would be generated once the `PPBytePair` type was designated as an exploded type, as there would then be both an exploded and a non-exploded type that implement the `PrettyPrintable` interface.

In this alternative, exploded and non-exploded types cannot implement the same interface. This may cause problems when only certain types implementing a given interface are converted to an exploded type, or when it is not feasible to modify all classes that implement a given interface, such as when working with multiple code-bases or library routines.

Specialize and Wrap: This alternative allows for mixing of exploded and non-exploded types that have been cast to a common interface type through the use of specialized code paths. In this example, separate versions of the `debuggingPrint` method would be generated, one that accepts exploded types that implement the `PrettyPrintable` interface and one that accepts non-exploded types implementing the interface.

In this alternative, exploded and non-exploded types can implement the same interfaces. Analyses must be used to determine when specialization is required. In cases where analyses are in-

```
33  Set<PrettyPrintable> s = new Set<PrettyPrintable>();
34  s.add(ppbp);
35  s.add(ppsp);
36
37  Iterator<PrettyPrintable> iter = s.iterator();
38  while ( iter.hasNext() ) {
39    PrettyPrintable pp = iter.next();
40    debuggingPrint(pp);
41  }
```

Figure 5.9: Example of mixing of exploded and non-exploded reference types through a common interface type.

sufficient[3] (e.g., when they can determine that mixing may occur, but not with enough precision to facilitate specialization), this alternative would wrap exploded object instances in Java objects as necessary, while providing compiler warnings indicating where costly wrapping will occur. This alternative may result in significant increases in code size due to code specialization, and may require additional memory and performance overhead in the form of run-time checks and run-time type information as well as potentially substantial overhead from wrapping exploded object instances.

With the above alternatives in mind, consider a more complex example. Suppose the additional code found in Figure 5.9 were also present. In some applications, this is a common pattern and thus merits discussion. The **Disallow** and **Independent** alternatives would still generate compiler errors as with the previous example, and are not of further interest. The **Specialize and Wrap** alternative would not be able to generate a straightforward specialized bit of code, as the `Iterator<PrettyPrintable>` class would need to be modified to accept a mix of reference types, but it would then be able to wrap instances of exploded objects (the `ppbp` variable, in this case) with a Java object to allow for both to be stored in the parameterized set. However, wrapping an exploded object in a Java object is an expensive prospect, and should generally be avoided if

[3]These analyses would be no less difficult than full alias analysis, which is an open problem, and a full solution to the general alias analysis problem is not appropriate as a portion of this work. Instead, it is plausible that analyses could be made to be "good enough" to catch the common cases, with certain rare cases being beyond the scope of the analyses.

possible.

For the special, and common, case of parameterized collections used in this somewhat stylized manner[4], one form of specialization may be appropriate for either of the specialization-based approaches. Specialized parameterized collections could be provided that are capable of storing references to both exploded and non-exploded objects. For example, a specialized `Set` could contain packed arrays of references to both exploded and non-exploded instances, with the addition of an array of indices into the two packed arrays indicating the traversal order for ordered collections. This approach would still result in considerable additional overhead for mixing of exploded and non-exploded types, but errs on the side of preserving existing code at the cost of performance.

For the purposes of the prototype implementation of the Restriction Approach, the **Disallow** approach has been taken.

The complexities and performance overhead introduced by the **Specialize and Wrap** approach hints at one of the motivations for the restrictions in the Restriction Approach, namely that allowing exploded object instances to be treated as subclasses of the Java `Object` class is difficult to do efficiently. Directly treating exploded objects as `Object` subclasses, such as would be the case if they were in the same type hierarchy, or indirectly treating them as subclasses, such as through casting to an interface type and then casting that interface type back to `Object`, requires supporting all of the features of Java's `Object` class and/or creating "objects" that do not share all of the semantics of Java `Object` subclasses— they may look like subclasses of `Object` but they don't act like them.

For example, supporting synchronization on an individual exploded object instance via synchronizing on a wrapper object (that is a real Java `Object` subclass) is not the same as synchronizing on the individual exploded object instance itself. Even if a thread has a lock on the wrapper object, other threads could modify the exploded object instance referenced by the wrapper, which violates the semantics of synchronization in Java. Likewise, supporting garbage collection of exploded object instances through a wrapper is not desirable, as there is no straightforward way in a standard Java virtual machine to match references (indices) to the exploded object instance being wrapped

[4]Even in cases where parameterized types are not directly supported, it is common to incorporate constraints on the types present in heterogeneous collections through run-time type checks, casts, or simple programming conventions such as comments indicating that "everything in this `Set` is a `PrettyPrintable`".

with the wrapper object itself; using a finalizer to explicitly delete the exploded object instance being wrapped when its wrapper is garbage-collected may result in dangling references, or conversely an explicit deletion of the exploded object instance being wrapped will result in dangling references in any wrapper objects of that instance. Object identity may become muddled, as well, where two different wrappers for one exploded object instance may be unequal at the pointer level (i.e., from comparison using ==), but may be equal at a higher level (e.g., through `.equals` or having identical hash codes), even when two exploded object references to the same exploded object instance will always be equal in every way.

The restrictions in the Restriction Approach are designed, in part, to avoid these and other complexities while still preserving a reasonable degree of flexibility, semantics similar to those of normal Java objects, and efficient internal representation and optimization opportunities.

Chapter 6

IMPLEMENTATION STRATEGY FOR THE RESTRICTION APPROACH

This chapter describes the implementation strategy used for the prototype implementation of the Restriction Approach to the exploded object translation process as described previously. Exploded object representation issues are presented first, along with motivation and examples. The source-to-source translation process employed by the implementation is presented next, followed by implementation concerns relating to casts. Analyses that are required for the implementation are presented next, and the chapter is concluded by a discussion of parameterized types as required by the prototype implementation.

6.1 Representation of Exploded Objects

Representing exploded objects has several aspects: collections of exploded objects (in which all instances are found), individual instances of an exploded object, and references to an exploded object instance, including cases where run-time type information is required.

6.1.1 Collections of Exploded Objects

There are several kinds of collections of exploded objects. The first is the canonical collection, a set in which all instances of the exploded object type are found. A canonical collection of exploded object instances is a Java object that contains four main parts: a collection of parallel arrays containing the fields of the exploded objects, a set of methods that correspond to the methods from the original exploded object definition, methods to implement the `Set<T>` and `FilterSet<T>` interfaces, and fields and methods to provide for creation and destruction of exploded object instances and to create the canonical set itself. Figures 6.1 and 6.2 contain definitions of the `Set<T>` and `FilterSet<T>` interfaces, which are provided as part of the prototype implementation and are designed to contain a reasonable subset of methods found in Java's `Set` interface. The canonical set wholly replaces the

original class, which is no longer present in any form after translation.

If the exploded object has a superclass, all field definitions from the superclass are duplicated in the subclass's canonical collection. All methods from the superclass are also copied, with appropriate variable modifications to ensure that references to instance variables refer to fields in the subclass's canonical collection. This is required to properly handle cases in which a subclass declares a duplicate field that shadows one declared in a superclass, and methods in each class refer to their separate version of the field. The normal Java protection mechanisms apply even though the fields and/or methods from the superclass are replicated, in that only replicated methods can refer to **`private`** methods or fields from the superclass.[1] **`static`** fields and methods from the superclass are not copied to the canonical set of the subclass, and references to **`static`** fields or methods of the superclass in the subclass remain as references to the canonical set of the superclass.

This strategy of replication combined with the normal Java semantics of dynamic dispatching (implemented via **`switch`** statements that direct a method dispatch into the appropriate canonical set; see 6.1.4 for more details) ensures that no references (index-based or otherwise) need to be maintained between the different canonical sets or across exploded object instances contained therein. If the canonical set for a subclass contained only the new fields added by the subclass, additional overhead would be introduced by requiring an instance of a subclass to contain a reference to a superclass instance that contains the fields defined in the superclass, and would require additional complexity in handling overridden methods as such methods may affect fields in multiple canonical sets (i.e., the subclass and one or more superclasses). The replication approach does not increase per-instance memory requirements needed to store fields. It does result in an increase in code size in the canonical sets due to the necessity to replicate method code from any superclasses, but this increase is generally modest for typical simulation entity objects whose methods are often no more complex than simple `get` or `set` accessors, and whose inheritance relations are generally very limited.

The class definition for the canonical collection of `Household` objects, with certain methods elided for brevity, is found in Figures 5.2 and 5.3. For the purpose of this example, it is assumed that the `Household` type has no superclasses or subclasses, so no run-time type information ever

[1]The generated code from the translation process may relax the access restrictions on fields in some cases, but type-checking prior to code generation ensures that the original restrictions are obeyed.

needs to be associated with a reference to a `Household`. If that were not the case, there would be an additional version of each of the methods for the `Set<Household>` and `FilterSet<Household>` interfaces that take a `Household` as a parameter with an additional argument for the run-time type. See Section 6.1.4 for more details. Lines 2–5 contain the parallel arrays, one for each of the original fields. All **`static`** fields retain their same level of access protection (e.g., **`public`**, **`private`**), as do all non-static fields once converted into array form.

Methods corresponding to non-static methods from the exploded object are converted to methods that accept an extra parameter, an **`int`** representing the index of the instance, and all references to instance variables are converted to references to the parallel arrays. Static methods from the exploded object are unchanged. Lines 16–21 show the conversion of several of the methods from the `Household` class.

Lines 22–29 and 30–32 show the methods for the `Set<T>` and `FilterSet<T>` interfaces. Lines 34–47 show the methods for creating and destroying instances of exploded `Household` objects through the use of `create` method(s) which replace constructors, and a `destroy` method for destruction. Note that one `create` method is present per constructor in the original class definition. The code in each `create` method corresponds to the code from the matching constructor, with modifications to handle obtaining a new free index in the parallel arrays as its first line. Any constructor calls (e.g., **`super(...)`** or **`this(...)`**) within a constructor are replaced by the code from the appropriate `create` method from the superclass, with variable renaming as appropriate to avoid name conflicts and to ensure that references to fields or variables in the superclass are modified to refer to the equivalent replicated fields or methods in the subclass. The index of the newly-created exploded object instance is returned by the `create` method.

Lines 49–51 show the other methods and fields associated with a canonical collection of exploded objects. Note the static reference to the canonical set found on line 9 with the accompanying static initialization method `init` found on line 12 that is responsible for initializing the canonical set to a certain initial size. The `init` method must be invoked explicitly by the user, so that the initial size is set to a useful value for the application, as opposed to having initialization occur within a **`static`** block in the canonical set. This requirement is appropriate for the domains of interest as the number of simulation entities to be stored is almost always known at run-time, such as immediately prior to loading all simulation entities from a file or database, or prior to synthesizing them,

```
1 public interface Set<T> {
2   public boolean isEmpty();
3   public int size();
4   public boolean add(T eo);
5   public boolean contains(T eo);
6   public boolean remove(T eo);
7   public Iterator<T> iterator();
8 }
```

Figure 6.1: Parameterized `Set` interface definition. This version is prior to the full translation process, and is what is exposed to developers. The type parameters represented by `T` are replaced by the type name (e.g., `Household` during the template instantiation process.

```
1 public interface FilterSet<T> extends Set<T> {
2   public FilterSet<T> filter(Filter<T> filt);
3 }
4
5 public interface Filter<T> {
6   public boolean filter(T obj);
7 }
```

Figure 6.2: Parameterized `FilterSet` interface and `Filter` class definitions.

but may or may not be known when compiling. The current implementations of canonical sets and specialized data types are not optimized to handle a range of dynamic sizes, but if new versions are supplied, canonical set initialization could occur automatically inside a **`static`** block.

The second kind of collection of exploded objects is a collection that contains only references to exploded object instances. Such collections typically implement the `Set<T>` or `FilterSet<T>` interfaces, or other parameterized interfaces as appropriate for the type of collection, such as interfaces for lists, hash sets, etc. All of these collections store only references to exploded object instances found within the canonical set, and all such references are **`int`** values indicating an index into the appropriate canonical set, based on type. If the run-time type may vary from the static

type, additional run-time type information will be stored along with the index as described in Section 6.1.4, on page 64. For performance reasons, and to save memory, specialized versions of some commonly-used collections such as hash sets, maps, and lists that store only a Java **int** are provided.

The template-based collection approach can be used to add new types of specialized collections as needed, as described in Chapter 7. Note that several versions of templates for each collection are required, one that provides the programmer-level interface that refers to the exploded object type as if it were a normal Java type, and one (or two) that provide the implementation-level code that deals directly with storing **int**s for indices and **byte**s for run-time type information if appropriate. The first is designed to help in the translation process from normal Java objects to exploded objects, by allowing for application testing using specialized collections but without requiring the full translation process, and the others allow for optimized implementations designed to store indices or indices with run-time type information in an efficient fashion. The programmer-level version typically wraps a traditional Java collection, and the implementation-level versions will implement the same interface(s) as the programmer-level version but make explicit use of **int** and **byte** types. These primitive types will be used to represent indices and run-time type information, but the implementation-level code does not need to reflect that. For example, a specialized `HashSet<T>` programmer-level template provides typed wrappers to a normal Java `HashSet`, such as an `iterator` method that returns a typed iterator (`Iterator<T>`) object, a version of `add(T)` that invokes `add(Object)` in the underlying `HashSet`, etc. The implementation-level versions are written as collections with a similarly-parameterized interface but which store **int**s or **int**s and **byte**s in parallel. The template instantiation process ensures that the former can be replaced by the latter during final translation by replacing the type parameters with the appropriate names, and ensuring that method signatures match. Providing specialized collections designed to store primitives eliminates the overhead that would otherwise be required to wrap **int**s with `Integer` objects for storage in a normal Java collection.

6.1.2 Individual Exploded Objects

One instance of an exploded object is represented as an index of type **int** into a collection of parallel arrays, with the fields of the instance contained in the arrays. Given the definition of a `Household`

class as found in Figure 5.1, an instance may be represented with the index `5` (for example), with its fields contained at index `5` in each of the arrays representing the type. The diagram in Figure 2.2 illustrates this representation.

Read or write accesses to public instance fields are translated into direct array accesses. Method invocations are passed to the appropriate canonical collection using run-time type information associated with the reference to determine which canonical collection receives the call. Note that in some cases, it may be possible to inline the method calls, though this has not been done in the prototype implementation. Figure 5.4 shows the translation of a code snippet that uses exploded `Household` objects. Lines 1–4 show the original code, and lines 6–10 shows the translation into Java code that makes use of the canonical set as described above.

6.1.3 References to Exploded Objects

References to exploded objects that are passed as parameters or returned from methods are replaced by an `int` representing the index of the corresponding instance, or an index plus run-time type information as described in Section 6.1.4 if necessary. Accesses to the instance within the method's body are translated as described previously.

6.1.4 Run-Time Type Information

When an exploded object type inherits from other exploded object types, it may be necessary to associate run-time type information with each instance. Instances that require run-time type information are known as *typed references*, and those that do not require run-time type information as *untyped references* for the purposes of this work. The presence of run-time type information must be accounted for during the compilation, analysis, and code generation phases, to ensure that space is available to store the run-time type information during program execution, and by ensuring that code generated by the compiler makes proper use of the run-time type information for dynamic dispatching, ensuring that the run-time type information is associated with a reference when the reference is passed to or returned from a method call, etc. In the prototype implementation, a reference to an exploded object instance is assumed to be a typed reference if it is a reference to a type that has any subclasses; this straightforward approach requires inspecting the class declarations of all exploded

object types, but no other analyses. Classes with no subclasses, including **final** classes, do not require run-time type information. Generally, a typed reference is represented as a pair of values, one **int** for the index and one `RTTI` for the run-time type information. `RTTI` will be replaced with the smallest Java primitive data type required to encompass all exploded types encountered during translation, which for the purposes of the prototype implementation will be a **byte** which can accommodate up to 255 unique types.[2] In some cases, such as returning a typed reference from a method, the reference is represented by a tuple containing the run-time type of the instance along with its index, stored in a `Pair<int, RTTI>` parameterized type which is defined in Figure 6.3. When such a tuple is required, it is immediately broken apart into a separate pair of primitive fields for subsequent use so that it is not necessary to retain a large number of `Pair<int, RTTI>` objects in memory.

All accesses to fields or methods through the reference pass through a method which dynamically dispatches the original invocation or access so that it involves the correct exploded object instance; this is a generalization of the translation strategy presented in the previous chapter. A **switch** statement based on the run-time type information is used to direct the field access or method dispatch to the appropriate canonical set, as shown at the bottom of Figure 6.6 and described later in this chapter. Run-time type identifiers for all exploded types are stored in a common class (e.g., `RunTimeTypes`) that is automatically generated and imported as needed. In some cases, static analysis can eliminate the need to return a tuple, such as when the return type's run-time type is statically known. The reference returned from a canonical set's `create` method is one example. More sophisticated analyses are not presently incorporated into the prototype implementation.

An example serves to illustrate this process. Suppose we have a subclass of the `Household` class as defined in Figure 5.1 used to represent households that have a home-based business. This type of class extension through subclassing (versus modifying the original class) may be used in cases where it is assumed that relatively few representatives of the original class will require the additional information defined in the subclass. Subclassing helps minimize the amount of additional memory required, as compared to adding the field to all instances of the original class. Note that the subclass also overrides the `toString` method to produce a more appropriate textual representation of an

[2]One value is reserved for `UNKNOWN` for error-handling purposes.

```
1 public class Pair<A, B> {
2   public A first;
3   public B second;
4
5   public Pair<A, B>(A f, B s) {
6     first = f;
7     second = s;
8   }
9
10  public String toString() {
11    return new String("(" + first + ", " + second + ")");
12  }
13 }
```

Figure 6.3: The `Pair` parameterized class definition, used as lightweight storage for pairs of values.

`SelfEmployedHousehold` than would be provided by the original `toString` method from the `Household` superclass. The `SelfEmployedHousehold` class definition is found in Figure 6.4. Note that it assumes a `HomeBasedBusiness` class definition, omitted for brevity.

Suppose some module in the simulation contains the code snippet found in Figure 6.5. Lines 3–15 populate the set of `Household` references defined in line 1, which is a parameterized set that holds references to exploded `Household` objects; the set can also contain references to `SelfEmployedHousehold` exploded objects, as the latter is a subclass of the former. As the `allHH` set contains references to both types of exploded objects, the call to `toString` on line 20 must be dispatched dynamically to ensure that the correct version of the `toString` method is invoked. This necessitates that adequate run-time type information be stored in the `allHH` set.[3]

Figure 6.6 shows how the code snippet from Figure 6.5 is translated. The code in the figure has been fully translated, so parameterized types have been replaced with normal Java types

[3]While analyses could reveal that the `allHH` set may require storing run-time type information (or always will, in this particular case), all collections in the prototype implementation that store typed references, will always include run-time type information. The sole exception is the canonical set, as all elements of the canonical set have a single type which is known statically.

```
1 public exploded class SelfEmployedHousehold extends Household {
2   private HomeBasedBusiness homeBasedBus;
3
4   public SelfEmployedHousehold() {
5     super();
6     homeBasedBus = null;
7   }
8   public HomeBasedBusiness getHBB() { return homeBasedBus; }
9   public void setHBB(HomeBasedBus hbb) { homeBasedBus = hbb; }
10  public String toString() { return new String("seHH#" + getID()
11    + ", hbb# "
12    + ((homeBasedBus != null) ? homeBasedBus.getID() : "[null]")); }
13 }
```

Figure 6.4: The `SelfEmployedHousehold` class definition, used to store households that run a home-based business.

that are template instantiations. Line 1 declares a `HashSet` template that has been instantiated to store what were `Household` references as its key; as the `Household` class has a subclass (the `SelfEmployedHousehold` class), the prototype implementation conservatively assumes that all `Household` references are typed references. The `HashSet_Household` type is thus a `HashSet` optimized to store an **`int`** index with a **`byte`** run-time type field in conjunction in an efficient fashion, through parallel arrays for the prototype implementation. In particular, it does **not** need to store one `Pair<`**`int, byte`**`>` object per typed reference, as doing so would eliminate any advantages to the exploded object representation of instances. This example assumes that there are no more than 255 different exploded types, so a **`byte`** would be of sufficient size to represent them all. In the rare cases with a greater number of unique exploded types, a **`short`** or **`int`** could be used. Alternatively, a bit vector could be used in parallel with the index set to use only the minimum memory needed to represent run-time type information, though at the cost of some performance.

Lines 2–14 populate the set of `Households` by iterating over a set of just `Households`, and a set of just `SelfEmployedHouseholds`. The first set (of externally employed `Households`,

```
1 HashSet<Household> allHH = new HashSet<Household>();
2
3 Set<Household> externallyEmployedHH = ...
4 Iterator<Household> hhIter = externallyEmployedHH.iterator();
5 while ( hhIter.hasNext() ) {
6   Household hh = hhIter.next();
7   allHH.add(hh);
8 }
9
10 Set<SelfEmployedHousehold> selfEmployedHH = ...
11 Iterator<SelfEmployedHousehold> seHHIter = selfEmployedHH.iterator();
12 while ( seHHIter.hasNext() ) {
13   SelfEmployedHousehold seHH = seHHIter.next();
14   allHH.add(seHH);
15 }
16
17 hhIter = allHH.iterator();
18 while ( hhIter.hasNext() ) {
19   Household hh = hhIter.next();
20   System.out.println(hh.toString());
21 }
```

Figure 6.5: Usage of references to `Households` and `SelfEmployedHouseholds` which requires run-time type information.

```
1 HashSet_Household allHH = new HashSet_Household();
2 Set_Household externallyEmployedHH = ...
3 Iterator_Household hhIter = externallyEmployedHH.iterator();
4 while ( hhIter.hasNext() ) {
5   PairIntByte hh = hhIter.next();
6   int hh_idx = hh.first; byte hh_rtt = hh.second;
7   allHH.add(hh_idx, hh_rtt);
8 }
9 Set_SelfEmployedHousehold selfEmployedHH = ...
10 Iterator_SelfEmployedHousehold seHHIter = selfEmployedHH.iterator();
11 while ( seHHIter.hasNext() ) {
12   int seHH_idx = seHHIter.next();
13   allHH.add(seHH_idx, RunTimeTypes.SELFEMPLOYEDHOUSEHOLD_TYPE);
14 }
15 hhIter = allHH.iterator();
16 while ( hhIter.hasNext() ) {
17   PairIntByte hh = hhIter.next();
18   int hh_idx = hh.first; byte hh_rtt = hh.second;
19   String tmp$0 = null;
20   switch ( hh_rtt) {
21     case RunTimeTypes.HOUSEHOLD_TYPE:
22       tmp$0 = hhCSet.toString(hh_idx); break;
23     case RunTimeTypes.SELFEMPLOYEDHOUSEHOLD_TYPE:
24       tmp$0 = seHHCSet.toString(hh_idx); break;
25     default: throw new RuntimeException(...);
26   }
27   System.out.println(tmp$0);
28 }
```

Figure 6.6: Translation of code that requires run-time type information to provide for proper dynamic dispatching, from the code in Figure 6.5. Note that the specialized collections have been fully translated through template instantiation to provide efficient storage of primitive types including run-time type information, and that the above code is produced by the automatic translation process.

from line 2) is likely to contain only `Household` references and no `SelfEmployedHousehold` references, but as the prototype implementation assumes that all `Household` references are typed references, the `someSelfemployedHH` set must contain run-time type information. Additionally, the iterator specialized for `Household`s must support returning both an index (**`int`**) and run-time type information (**`byte`**), so it returns a `Pair<int, byte>` (line 5, shown in post-translation `PairIntByte` form) which is picked apart into its index and run-time type portions (line 6) before being used.

Note that while iterating over a set of `SelfEmployedHousehold`s, no run-time type information is needed as `SelfEmployedHousehold` has no subclasses. The run-time type information needed for the `add` method for the `Set<Household> allHH` (shown in its post-translation form as `Set_Household`) is statically known and is filled in appropriately, as shown in line 13.

Once the `allHH` set has been populated with a mixture of `Household` and `SelfEmployed-Household` references, the loop in lines 15–28 iterates over the set and prints each one in turn. Line 18 show the `Pair<int, byte>` (again in post-translation `PairIntByte` form) being returned from the iterator and broken into its index and run-time type information parts. Lines 19–26 show the translation of the dynamic dispatching from the original `toString()` invocation. A temporary variable is declared to store the result, and a **`switch`** statement is inserted that is based on the run-time type of the current `Household` as returned by the iterator. The `toString()` method from the appropriate canonical set is invoked based on the run-time type of the current `Household`, and the result of that invocation is printed in line 27.

6.2 Source-to-Source Translation

The prototype implementation employs a source-to-source translation strategy, generating normal Java code as output. This approach was chosen for three reasons: to maximize compatibility with standard Java virtual machines, to leverage future developments in Java compiler technology, and to facilitate testing and development.

One of the requirements of the restriction approach is that it function with standard Java virtual machines. By generating normal Java source code, any Java compiler can be used to generate Java bytecode suitable for any particular JVM. While it is likely that a prototype that generated bytecodes

would still allow for a high degree of compatibility with a range of virtual machines, it seems more likely that the binary format of Java bytecode will undergo changes than the underlying language would be changed to the extent that source-code generation would no longer be compatible with future versions of Java compilers.

Java compiler technology is likely to improve in the future. By generating standard Java source code, the prototype implementation allows for maximal leveraging of future Java compiler improvements. Not all Java compilers can accept Java bytecodes as input, whereas Java source code should be universally accepted. The prototype can also benefit from bytecode-to-bytecode optimizers by first compiling the generated source code with any Java compiler and then feeding the resulting bytecode to the bytecode optimizer.

From a practical standpoint, it is much easier to produce Java source code as output of the prototype given that the prototype is built upon the Polyglot [33] extensible compiler framework, which provides a simple mechanism to generate Java source code from internal compiler data structures but does not include a mechanism to directly produce Java bytecode. Additionally, it is significantly easier to manually analyze and test the output of a prototype implementation that generates Java source code as compared to disassembling Java bytecode and attempting to match that back to the original Java source input.

6.3 Casts of Exploded Object Instances

As with normal Java objects, exploded object instances can be upcast (cast to a direct or indirect superclass) and downcast (cast to a direct or indirect subclass), but in the prototype implementation cannot be cast to interface types as exploded object types cannot implement interfaces.

From an implementation standpoint, an upcast requires a compile-time check to ensure that the upcast is legal, based on the static type of the exploded object instance and the cast type, but otherwise has no effect and causes no code to be generated.

A downcast requires a run-time check to ensure that the downcast is legal, based on the run-time type of the exploded object instance, but has no other effect; even in the absence of downcasts, dynamic dispatching ensures that the most specific method is invoked, or that the most specific field (as stored in the appropriate array in the relevant canonical set) is accessed. A helper method which

encapsulates all legal downcast relationships is used to verify that the downcast is legal, and throws a `ClassCastException` if it is not.

6.4 Required Analyses

In order to perform the translation of exploded objects as described in this work, and to facilitate certain straightforward extensions to the prototype implementation, several forms of analysis are required.

6.4.1 *Inheritance Relationship Analysis*

To ensure that no exploded object type inherits from a non-exploded type, or vice-versa, a simple local check is required to examine the **`extends`** portion of each class declaration. To ensure that the rules regarding implementation of the same interface by exploded and non-exploded types are not violated, the **`implements`** portion of each user-defined type (or the equivalent extracted from types found only in libraries) is examined. A compiler error is generated if any interfaces are implemented by both an exploded and a non-exploded type.

6.4.2 *Run-Time Type Analysis*

For the purposes of the prototype implementation, analyses to determine possible run-time types of variables is not strictly necessary. The prototype implementation makes a conservative assumption that all typed references always require run-time type information, and translation and template instantiation is performed accordingly. This assumption can negatively impact performance and memory usage by requiring support for run-time type information even when it may not strictly be necessary. For example, the set of externally-employed `Household`s declared in line 2 in Figure 6.6 is homogenous, storing only `Household` references and never references to `Household` subclasses. However, the prototype implementation will instantiate the `Set<Household>` template using the version that stores run-time type information, and every run-time type field in the set will be identical.

Alias and flow analysis could improve upon the assumption made by the prototype implementation, resulting in better selection of which specialized type instantiation and typed/untyped reference

to use in any given context. For example, if alias analysis can determine that all exploded objects in a specialized collection are of the same run-time type, or that an exploded object passed as a parameter or returned from a method has a fixed run-time type, specialized code could be generated to avoid the overhead of dealing with dynamic manipulation of run-time type information. Likewise, data flow analyses could be used to determine how a `Pair<int, byte>` representing a reference to an exploded object with run-time type information was later used, to determine if and when it should be broken apart into separate fields for the index and run-time type information.

Rudimentary alias analysis would be required to extend the prototype implementation to allow for interfaces to be implemented by both exploded and non-exploded types. For example, it would be needed to determine whether a given reference (or collection of references) to an interface type implemented by both exploded and non-exploded object types may ever share the two types or not. A simple and highly-conservative analysis should suffice, as it is always safe to return "yes, it may be shared" and translate accordingly for any given reference, though limiting specialization and translation to cases only where the references are definitely shared can help reduce code size and may improve performance. An overly conservative analysis would prevent direct re-use of existing code, as well, by requiring all existing code to be converted to the possibly-shared version.

Data flow analyses would be required to determine scoping of new variables created for translations of mixed-reference variables. They may need to be bi-directional (forward or backward) in nature. Control flow analyses are needed to determine a variety of information, and may be needed to support other analyses. Both of these are well-known analyses, and should be reasonably straightforward to add to the prototype implementation if desired.

6.5 Parameterized Types

In order to provide efficient representation of collections of exploded object references, parameterized types that support heterogeneous translation must be present. Collections of primitive types, such as `Set<int>`, are used extensively by the generated code to support the Java programming idioms of collections and iteration, and are required for certain aspects of the canonical sets such as free list management. It is acceptable to have no support for nested parameterized types, which means that Centralia-style parameterized types [31] are sufficient. A template-based approach to

parameterized types has been implemented as part of the prototype implementation, as described in Chapter 7.

Some care must be taken in selecting a mechanism for supporting parameterized types, as specialized types which store typed references must be translated differently than those storing untyped references. For example, if a `Household` type has `SelfEmployedHousehold` as a subclass and the latter class has no subclasses of its own, a `Set<Household>` may require a different instantiation than a `Set<SelfEmployedHousehold`. The former `Set` requires (or may require) support for storing run-time type information along with each index, whereas the latter `Set` will not. Furthermore, the `Set<Household>` instantiation must be efficient in its storage of typed references; storing an entire Java object (e.g., a `PairIntByte`) per typed reference is unacceptable as doing so results in one Java object instance per exploded object instance. The template-based approach to parameterized types as described in Chapter 7 is designed to address exactly these issues.

Chapter 7

TEMPLATE-BASED SUPPORT FOR SPECIALIZED COLLECTIONS

A version of parameterized types based on templates has been implemented as part of the prototype system in order to provide support for specialized collections and data types involving exploded object types. Templates are used to define specialized data types, and the templates are instantiated to generate normal Java source code as part of the source-to-source translation process of the prototype implementation.

Each template supports a number of tags which are replaced by textual substitution or generated code, as described in Table 7.1. The template system facilitates the addition of new types of specialized data types or collections, as a user of the system can create a small number of templates which are then instantiated as fully-specialized data types. The template system can also be used to experiment with tuning collections for specific applications, as different specialized variants of a collection can be created as easily as one would create a specialized variant for ordinary Java code.

7.1 Template Categories

There are three different categories of specialized collection templates, as described below.

7.1.1 Interface Templates

Interface templates are used for parameterized interfaces. Examples include interfaces for specialized sets, lists, and maps, such as the specialized `Set` template shown in Figure 7.1.

7.1.2 Intermediate Templates

Intermediate templates are used for generating specialized types in the intermediate generation phase of the translation process. They are generally wrapped versions of normal Java collections, such as `java.util.HashSet` and `java.util.ArrayList`. Figure 7.2 shows an intermediate version of

Table 7.1: Template tags for use in specialized parameterized type templates and their effects.

Tag	Effect
<BANNER>	Inserts a comment banner into a block (/* */) comment.
<PACKAGE>	Inserts a package designation for the specialized data type.
<COMMON_IMPORT>	Inserts **import** statement(s) for classes commonly needed by generated types.
<T>	Used to specify the return type of a method. The tag is replaced by **int** for exploded types that do not require run-time type information, or Pair<**int**, **byte**> for ones that do. The prototype implementation assumes ≤ 255 distinct exploded types.
<TYPENAME>	The tag is replaced by the name of the type. For example, List_<TYPENAME> is replaced by List_Household when instantiating the template for the Household exploded type.
<T=*o*>	Used to specify the name of a parameter whose type is the specialized type. The tag is replaced by **int** o_idx for exploded types that do not require run-time type information, or **int** o_idx, **byte** o_rtt for types that do.

Table 7.2: Template tags used for canonical set instantiation.

Tag	Effect
<CREATE>	This tag is replaced by the `create()` methods for the canonical set, with one `create()` method generated for each separate constructor in the original exploded object type.
<FIELDS>	This tag is replaced by parallel arrays that contain the fields of the original exploded object and all fields from all superclasses. For example, the **byte** `age` field of a `Household` would result in the generation of **byte[] age**.
<FIELDINITS= *initSz,ref*>	This tag is replaced by a list of field initializers that initialize the arrays representing each field by allocating new arrays of size *initSz*. The reference through which the arrays are accessed is specified by *ref*. For example, the **byte** `age` field of a `Household` would result in the generation of `ref.age = new byte[initSz]`.
<STATICINITS>	This tag is replaced by a block containing the static initializers of all static fields declared by the original exploded object type.
<FILTER-_INTERFACE>	This tag is replaced by the `filter` method required for the canonical set to implement the `FilterSet_<TYPENAME>` interface.
<SET_INTERFACE>	This tag is replaced by the method(s) required for the canonical set to implement the `SET_<TYPENAME>` interface. At present, this consists solely of iterator-based methods and local class definitions.
<GROWARRAY>	This tag is replaced by a method used to grow the parallel arrays when more space is required to store exploded object instances.
<DESTROY>	This tag is replaced by the `destroy` method used to destroy an instance of the exploded type.

Table 7.3: Additional template tags used for canonical set instantiation. Methods generated by each tag are modified to include an extra parameter to specify the index.

Tag	Effect
<TOSTRING>	This tag is replaced by a **public** `String toString` method if one was not provided in the original exploded type's definition. If one was provided, it is used to replace the tag.
<HASHCODE>	This tag is replaced by a **public int** `hashCode` method if one was not provided in the original exploded type's definition. If one was provided, it is used to replace the tag.
<EQUALS>	This tag is replaced by a **public boolean** `equals` method used to compare two instances of the exploded object type. If an `equals` method was provided in the original exploded object type, it is used to replace the tag. Otherwise, a default version is generated.
<METHODS>	This tag is replaced by all the methods from the exploded object's type, and all supertypes, except for those methods that would be used to replace another tag (e.g., `toString`, `equals`).

```
1 /**
2  * Template for Set interface used in specialized Sets of Exploded
3  * Objects.
4  *
5  * <BANNER>
6  */
7
8 <PACKAGE>
9
10 <COMMON_IMPORT>
11
12 public interface Set_<TYPENAME> {
13
14   public boolean add(<T=o>);
15   public void clear();
16   public boolean contains(<T=o>);
17   public boolean isEmpty();
18   public Iterator_<TYPENAME> iterator();
19   public boolean remove(<T=o>);
20   public int size();
21 }
```

Figure 7.1: Interface template for specialized `Set` interface.

a specialized `HashSet`. The middle portion of Figure 5.5 shows an example of making use of the intermediate form of a specialized `Set` template, as do Figures 5.9 and 6.5.

Instantiated versions of intermediate templates are used to help with the transition process from normal Java objects to exploded objects. They provide the same interface to the specialized type as is found in the final translated version that store indices, but do not require the full translation process to be used. This allows for testing a version of the original application that makes use of specialized types, including verifying that the semantics and interfaces of specialized collections are taken into account if they differ from their Java collection counterpart, without requiring the added complexity of performing the full translation process. The intermediate forms of templates also provide for effective usage of the underlying Polyglot compiler infrastructure by allowing more aggressive use of its type-checking visitors than would be possible in a single-pass approach. Appendix A describes the use of the Polyglot extensible Java compiler as the basis for the prototype implementation.

7.1.3 Final Implementation Templates

Two forms of templates are required for the final, fully-translated version of a specialized type, one that assumes the type requires run-time type information, and one that does not. The appropriate template is selected for generation based on each exploded type that is present. Examples include templates similar to Java's `HashSet` and `ArrayList` collections.

The final implementation templates are versions of the templates that store actual **`int`**s and **`int`**s plus **`byte`**s in combination, and are used to store untyped references and typed references respectively. The template instantiation process, combined with the rest of the translation process performed by the compiler, ensure that the instantiated final form of a template can be used to replace its intermediate form. Figures 7.3 and 7.4 show a portion of the final implementation form of a specialized `HashSet` collection that is designed to store typed references consisting of **`int`** indices and **`byte`** run-time type information. Note that the template stores **`int`** and **`byte`** values in conjunction in parallel arrays, and not as pairs in a Java object, to avoid having one Java object per stored reference. Likewise, the iterator returned from the template only stores one internal `Pair<`**`int`**`, `**`byte`**`>` which is returned by the `next()` method of the iterator. By updating its fields while stepping through the set, it is not necessary to have one `Pair<`**`int`**`, `**`byte`**`>` per stored

```
1 /**
2  * <BANNER>
3  */
4 <PACKAGE>
5 import java.util.*;
6
7 public class HashSet_<TYPENAME> implements Set_<TYPENAME> {
8   protected HashSet objSet;
9
10  public HashSet_<TYPENAME>() { objSet = new HashSet(); }
11  public HashSet_<TYPENAME>(int initSz) { objSet
12    = new HashSet(initSz); }
13
14  public boolean add(<T=o>) { return objSet.add(o); }
15  public void clear() { objSet.clear(); }
16  public boolean contains(<T=o>) { return objSet.contains(o); }
17  public boolean isEmpty() { return objSet.isEmpty(); }
18  public Iterator_<TYPENAME> iterator() {
19    return new HS_Iterator_<TYPENAME>();
20  }
21  public boolean remove(<T=o>) { return objSet.remove(o); }
22  public int size() { return objSet.size(); }
23
24  class HS_Iterator_<TYPENAME> implements Iterator_<TYPENAME> {
25    protected Iterator iter;
26    public HS_Iterator_<TYPENAME>() { iter = objSet.iterator(); }
27    public boolean hasNext() { return iter.hasNext(); }
28    public <T> next() { return (<T>) iter.next(); }
29  }
30 }
```

Figure 7.2: Template for the intermediate version of a specialized `HashSet` class. Note that it acts as a wrapper for the normal Java `HashSet` collection.

reference. The translation process will immediately break the returned `Pair<int, byte>` into separate fields and never maintain a persistent reference to it.

7.2 Canonical Set Templates

In addition to specialized collection templates, two templates are provided for use in generation of the canonical sets in which all instances of a given exploded object type are found. One version is the intermediate form, which provides the same interface to the collection of exploded object instances as the final translated version provides, with matching semantics, and the other is for the final, fully-translated version. Canonical set templates support some additional tags, as described in Tables 7.2 and 7.3. The intermediate form is provided to help users ensure that they have properly modified their application's usage of exploded object types to match the semantics of exploded objects and their specialized collections. It allows users to compile and execute the modified version of their application using data structures that reflect the semantics of exploded objects. The intermediate form also simplifies several parts of the prototype's implementation by allowing it to use a bootstrapping process to facilitate parsing and analysis of the use of specialized collections. The final version is used with the fully-translated version of the application.

The processing of tags from Tables 7.2 and 7.3 is more complex than simple textual substitution as is used for the tags in Table 7.1. Several of the tags, such as `<CREATE>` and `<EQUALS>`, involve extracting code from the original exploded object class, modifying the code to work with object instances that are split apart into parallel arrays, and ensuring that duplicate names are appropriately mangled so they can be differentiated. For example, expanding the `<METHODS>` tag for an exploded object type that has a supertype involves extracting all the methods of the type in question, modifying each field access within each method body to index into the appropriate parallel array in the canonical set, and doing the same process for all methods from the superclass while ensuring that any field references from within the methods of the superclass refer to fields declared in the superclass and not the subclass if the subclass declared new fields with the same name.

```
1  /* Set for storing index, run-time type info in combination
2   * <BANNER>
3   */
4
5  <PACKAGE>
6  import java.util.BitVector;
7  <COMMON_IMPORT>
8
9  public class HashSet_<TYPENAME> implements Set_<TYPENAME> {
10   ...
11   // The array of keys
12   private int[] keys_idx; private byte[] keys_rtt;
13   ...
14   public boolean add(int key_idx, byte key_rtt) {
15     // Invalidate iterators
16     ...
17     int numProbes = 0, newIdx;
18     while ( true ) {
19       newIdx = getHash(key_idx, key_rtt, numProbes);
20       ...
21       if ( ((keys_idx[newIdx] == NULL_KEY)
22            && (keys_rtt[newIdx] == NULL_KEY_RTT)) || testBit(newIdx) ) {
23         // Found an empty space, store the value here
24         keys_idx[newIdx] = key_idx; keys_rtt[newIdx] = key_rtt;
25       ...
26     } ...
27   }
28   public boolean contains(int key_idx, byte key_rtt) { ... }
```

Figure 7.3: Part one of excerpts from a template for the final implementation version of a specialized `HashSet` class that stores index plus run-time type information. Note that methods such as `add` take two parameters, an **`int`** as an index and a **`byte`** for run-time type information.

```
29  public Iterator_<TYPENAME> iterator() {
30    return new HashSet_<TYPENAME>.KeyIterator();
31  }
32  ...
33  // Inner iterator class
34  public class KeyIterator implements Iterator_<TYPENAME> {
35    private PairIntByte iterPIB;
36
37    public KeyIterator() { ... }
38    public boolean hasNext() { ... }
39    public PairIntByte next() {
40      int retval_idx = keys_idx[keyIterIdx];
41      byte retval_rtt = keys_rtt[keyIterIdx];
42      ...
43      iterPIB.first = retval_idx; iterPIB.second = retval_rtt;
44      return iterPIB;
45    }
46  }
47 }
```

Figure 7.4: Second part of partial template for the final implementation version of a specialized `HashSet` class that stores index plus run-time type information. Note that the iterator returned by the template only returns a single `Pair<int, byte>` object instead of needing to create one `Pair<int, byte>` per reference in the set.

Chapter 8

EXPERIMENTAL EVALUATION OF THE PROTOTYPE IMPLEMENTATION

In order to evaluate the effectiveness of the Restriction Approach as implemented in the prototype implementation, a set of applications and tests were written in Java and then converted to use exploded objects. Memory usage and running time were compared between the original Java version and the translated exploded object version. The results demonstrate to what extent the prototype implementation meets the performance goals of doubling performance and halving memory requirements. This chapter describes the experimental configuration used to run the experiments, the set of applications used in the evaluation, and presents the results of the experiments. An evaluation of the results, along with an evaluation of the usability of the prototype implementation, round out the chapter.

8.1 Experimental Configuration

All experiments described in this chapter were executed on a dual-processor 1 GHz P-III machine with 1.5 GB RAM running Windows 2000. Only one processor was used for the experiments, and the machine was otherwise idle during experiment execution. The Java runtime environment used Sun's Java 2 Runtime Environment, Standard Edition, build 1.4.1_01-b01, with the HotSpot JIT enabled and executing in mixed interpreted/native mode. JVM heap size was specified using the `-Xms` and `-Xmx` command-line to pre-allocate the heap size to the full amount used for each experiment.

Timing results were obtained by inserting calls to the Java system utility method `System.currentTimeMillis()` before and after top-level method calls that perform the computation within each application being evaluated. Timing results do not include overhead for JVM initialization or class loading, which should be roughly comparable across all experiments. Compilation/translation times do not factor into the reported experimental results. Each experiment was run three times and

the average reported time was recorded.

To compare the performance of the translated exploded object version of each application with the original version that used only Java `Objects`, the relative performance increase (or decrease) was computed by dividing the average reported time of the original version by the average reported time of the version using exploded objects. Performance factors greater than 1.0 indicate that the exploded object version was faster, with a value of 2.0 or greater indicating that the desired performance goal of doubling performance was met.

Memory results were obtained by decreasing the heap size allocated to both the original Java object version and the translated exploded object version of each application until the application would fail with an `OutOfMemoryError`. The smallest heap size at which the application did not fail was then recorded. This approach was deemed to provide an "apples to apples" comparison in that it measures memory requirements including all required infrastructure for both versions. For example, collections used to store references to objects, including wasted space due to inefficient collections, overhead due to the virtual machine, code size, overhead due to the garbage collector, etc., are all a factor in evaluating the effective memory usage of an application.

8.2 Description of Experimental Applications

To evaluate the performance of exploded objects as realized in the prototype implementation, three test applications were created: a synthetic benchmark that is designed to exercise specific tasks common to simulation environments in which exploded objects may be of use, an "UrbanSimlet" that contains several model components that are based on actual portions of UrbanSim, and a data synthesis tool used to generate test data for use by the UrbanSimlet. Each application was tested in its original form that relied on only normal Java objects and collections such as `java.util.HashSet`. One or more of the classes within the application were then tagged with the **`exploded`** keyword, code involving those types was modified to make use of specialized data types provided during the intermediate stages of the exploded object translation process, and then the fully-translated version making use of specialized data types was executed to compare with the original version. Each of the three test applications is described in more detail below.

The test applications were designed based on the author's experience implementing UrbanSim

model components and are intended to be reflective of typical UrbanSim-style simulation computations. Normal Java idioms, such as iteration over collections which is the typical idiom found in UrbanSim, were used.

A collection of microbenchmarks was also written to help evaluate the cost and efficiency of different aspects of the Restriction Approach as realized in the prototype implementation.

8.2.1 Synthetic Benchmark Application

The synthetic benchmark application is loosely based on an UrbanSim-like simulation environment, where simulation entities include businesses, households, people, etc., with inheritance being used in several cases. For experimental purposes, all simulation entity types were tagged as being exploded. The benchmark is divided into three main portions: an initialization stage that allocates objects, a pass that iterates over all people (`Person` and its subclasses) and extracts some statistics, and a pass that computes the internal ID range of simulation objects.

The initialization stage creates a number of different types of simulation entities and stores them in specialized collections. Collections used to store objects were pre-initialized with the number of objects to be stored, as is typical in simulation environments where the number of simulation entities is known prior to execution. The initialization stage makes heavy use of object allocation (via **new** for the original Java object version, and via the canonical set's `create` method for the translated version) and specialized data types.

The statistic-gathering pass iterates over all `Person` instances (but not its subclasses, such as `Employee` instances) as found in the application. Two methods are invoked on each `Person` instance, to extract the education level and gender, and the method results are aggregated for later reporting. Even though no subclasses of `Person` are in the set of instances iterated over, dynamic dispatching is still required as subclasses of the `Person` type exist. This pass makes use of method invocation and dynamic dispatching, as well as iteration over a specialized collection.

The pass that computes the internal ID range of simulation objects first adds all simulation objects of each type into a single set, then iterates over the set of all simulation objects and invokes a method on each one to extract its internal ID. This process simulates typical minimum- and maximum-extraction procedures, such as those that may be used to determine income range of a

set of households, in an environment where dynamic dispatching is required. Separate runs were performed to evaluate the effects of initializing the size of the set into which all simulation objects were added as compared to using the default set size. The minimum and maximum range of internal ID values is then reported. The pass makes heavy use of specialized collections and dynamic dispatching. All simulation objects inherit from a `SimObj` base class, so run-time type information is associated with every instance at every step of the computations in this pass. The effects of the pass are to test the specialized collections, run-time type information, and dynamic dispatching.

8.2.2 UrbanSimlet Application

The UrbanSimlet application is structured much like UrbanSim, though in a somewhat simplified form. It contains an in-memory collection of simulation entity objects, a group of model components that represent distinct behaviors in a simulation, and an output stage that stores simulation results to disk. The UrbanSimlet written to test the prototype implementation of exploded objects represents a portion of the UrbanSim simulation involving residences (households). It contains three model components, a demographic transition model that creates or destroys households to match exogenous population totals for the simulation, a residential mobility model that determines which households will attempt to relocate during each simulation year, and a residential location choice component that determines where unplaced households will attempt to locate. Land is represented using a grid of grid cell objects that contain total quantities of housing units in that cell, retail or other forms of employment, accessibility values indicating the degree to which employment can be accessed from the grid cell, etc. Households consist of an income category level, an age, a size, a flag indicating if they have children, and a grid cell location indicating where their residence is located.

The loading phase of the UrbanSimlet involves creating simulation objects for households and grid cells and populating them with data loaded from external files. The original version of the application uses normal Java `HashSets` to store simulation object instances, and the translated version stores instances within canonical sets. This phase of the simulation exercises object creation and the translation of constructors as found in the `create` methods within the canonical sets in the translated version of the application.

The demographic transition model compares the current total population level with exogenous population totals for the current simulation year. If the current population total is below the target total, new households are created and added to the simulation environment until the target population total is met. If the current population total is greater than the target, randomly-selected households are deleted and removed from the simulation until the population has been reduced to the target level. When households are removed, households with no location are deleted before any placed households are deleted.

The residential mobility model examines every placed household in the simulation and determines whether that household will leave its current residence to try to find a new location or will remain in place. Relocation probabilities are based on the household's attributes. Households that choose to relocate become unplaced, with the residential location choice model attempting to place them later in that simulation year.

The residential location choice model attempts to place every household that has no residence (i.e., is unplaced) into a grid cell that has at least one available vacant housing unit. First, the model computes the set of every possible grid cell into which a household may locate, which is the set of all grid cells with at least one vacant housing unit. Then, each unplaced household randomly selects a fixed number of alternatives (set to 20 during experimentation) from the set of possible locations, and ranks each one with relative desirability based on household and grid cell attributes. Once every unplaced household has ranked its set of alternatives, a second pass is made over all unplaced households. Each household is placed into its highest-ranked alternative location that still has at least one vacant housing unit. If a household cannot be placed in any of its alternatives (e.g., if there are fewer vacant housing units than there are unplaced households, and other households have already been placed into and filled up the current household's alternatives), it remains unplaced. This pass involves substantial use of specialized collections, as well as creation and deletion of large numbers of temporary objects that are used to store and order the location choice alternatives of each household.

The output stage of the simulation writes the current state of all household and grid cell simulation objects to external files. This entails iterating over each object and invoking accessor methods to read its fields, though it is dominated by disk accesses.

Execution of the UrbanSimlet consists of loading the data, running the three model components

once each simulation year for a number of simulation years, and then producing output of the state of the simulation objects. For experimental purposes, one million households were created, along with a grid of grid cells measuring one thousand by one thousand grid cells for a total of one million grid cells. Target population totals were supplied so that the total population would rise from roughly 3.5 million people (or roughly one million households) to 4.0 million people (or roughly 1.15 million households) over a five year simulation run. Enough housing units were present in the grid cells so that all households could be placed.

For experimental purposes, separate runs were performed with the UrbanSimlet where the class object used to store the pairs of location and ranking information used for the residential location choice model was a normal Java Object or was converted into an exploded object. Households and grid cells were represented as exploded objects for all runs of the translated version.

8.2.3 Data Synthesis Application

To provide input data for the UrbanSimlet application, a data synthesis application has been written. It generates grid cell and household objects based on hard-coded attribute distributions, places each household into a grid cell with at least one vacant housing unit, computes accessibility values, and saves the data. Grid cells are stored in a two-dimensional array and households are stored in a set in the original version. The translated version uses the canonical set for household objects to store household instances, with grid cell instances stored in the grid cell's canonical set and a two-dimensional array of grid cell references being maintained in parallel.

Household and grid cell generation involves generating attributes for the object using predetermined probability distributions, then instantiating an instance of the object using the randomly-determined attributes. These processes consist of object creation, via **new** for the original Java Object version, or via the `create` methods from the appropriate canonical set for the exploded object version.

Placing household instances in grid cells entails examining all grid cells to build a set of candidates consisting of every grid cell that has at least one vacant housing unit. Each candidate grid cell is then filled with unplaced households by placing unplaced households into it until no vacant housing units remain. This process entails using specialized data structures to store sets of candi-

dates, iteration over the same, and accessing methods in simulation object instances to get and set field values.

Accessibility computations entail stepping over each grid cell in the two-dimensional array of grid cells and invoking a helper method to compute accessibility values for that grid cell. Accessibility to retail and employment is computed by summing the contributions from each cell in a certain radius around the target cell (set to 5 grid cells away for experimental purposes), with the contribution decreasing by distance according to an inverse square law. The log of total contributions is stored as the final accessibility value. This process involves many calls to grid cell object methods to get and set values.

Saving the generated data consists of iterating over each household and grid cell, fetching their attributes through calls to accessor methods, then writing the values to files. This process emphasizes iteration and invocation of methods on simulation object instances, though it is dominated by disk accesses.

8.2.4 Microbenchmarks

A collection of microbenchmarks have been written to measure the relative cost of certain operations under the translation strategy used by the Restriction Approach as compared to the equivalent operation in Java, or to measure the absolute cost when no Java equivalent is available.

Three groups of microbenchmarks have been evaluated. The first group compares cost of reading an instance field, writing an instance field, and invoking an instance method that does not return a result, with separate evaluations of conditions where no inheritance is present at all (the class has no subclasses or superclasses, apart from `Object` for the Java version), and where the reference has a static type of a superclass and a dynamic type of a subclass to force a dynamic dispatch. For these experiments, no difference was measured for **`final`** or non-**`final`** classes and/or methods in either the normal Java version or the exploded object version. The latter case evaluates the effectiveness of the translation strategy at handling dynamic dispatching and accessing fields that may be shadowed; one would expect field reads and writes to behave identically under the Java version, whereas their translations in the exploded object version differ due to altered code cache behaviors based on which case in the `switch` statement is executed, costs of branch prediction failures in the `JIT`-optimized

code, etc.

The second group of microbenchmarks measures the cost of object creation by comparing the cost of Java's **new** to the cost of `create` in the canonical sets. Two sets of evaluations have been performed, one for a clean environment in which no other allocation has taken place, the other in a "dirty" environment where a large number of instances are allocated and half of them then deallocated prior to the timed allocations. These microbenchmarks are designed to measure the effect of internal fragmentation on allocation performance for the canonical sets as compared to purely sequential allocation.

The third microbenchmark measures the absolute time required to destroy an exploded object instance by calling the `destroy` method in its canonical set. Two sets of runs were performed, the first testing sequential deletion by allocating a large number of instances and then destroying them all, and the second testing random deletion by allocating a large number of instances, permuting the list of references to those instances, and then deleting every element in the permuted list.

Each microbenchmark is contained within a separate method in a test harness class. Each method contains some setup code, then a simple loop that contains one or two statements that exercise the feature being evaluated, with the loop being executed one hundred million times for the first group of microbenchmarks, and involving five million instances for the second and third group of microbenchmarks. To more fully account for the effects of the just-in-time compiler, the JIT was "primed" by executing each microbenchmark method once, using a smaller number of iterations, prior to the full-scale timed execution of the same method. To attempt to account for loop overhead, a loop with identical contents to the microbenchmark but missing the statement(s) being measured was executed twice as well, and the time required to execute the empty loop the second time was subtracted from the time required to run the loop including the microbenchmark statement(s). Each microbenchmark was executed three times with the average time being used.

8.3 Experimental Results

This section contains summaries of experimental results gathered by executing the three test applications as described above. Each graph of results plots the performance factor for different aspects of each application, including total execution time, as a ratio of average time for three runs of the orig-

inal Java object version compared to the average time for three runs of the fully-translated exploded object version.

Each set of results for the test applications is discussed below, accompanied with test application-specific observations and analyses regarding the results. General analysis of all results follows the analysis of the individual test applications.

As the results of the microbenchmarks can help explain the results of the test application experiments, the microbenchmark results are presented first.

8.3.1 Microbenchmark Results

Table 8.1 contains the results of the first group of microbenchmarks. Results are shown for instance field writes, field reads, and instance method calls, each listed with a class that does not make use of inheritance (no inh) and a class that makes use of inheritance (inh) where the reference has a static type of the base class and a dynamic type of a `final` subclass[1]. The cost per operation for the Java and exploded object versions, shown in nanoseconds, is found in the first two columns of numerical data. The rightmost column contains the factor by which the exploded object version was faster (for values greater than one) or slower.

Given that the exploded object translation approach introduces an additional level of indirection and an array access for every instance field access, it is not surprising that instance field reads and writes are faster for the Java version after the JIT has had an opportunity to optimize the code. In the presence of subsumption, the exploded object version requires a `switch` statement to ensure that the appropriate version of a possibly-shadowed field is accessed, which accounts for the decrease in performance for the field accesses involving inheritance and subsumption. The difference in time required for a field read versus a field write in the presence of subsumption is due to the current translation approach that introduces a temporary variable within the `switch` statement and uses it for the source of the assignment; manually removing the use of the temporary variable eliminates the difference between reading and writing for the exploded object version. A more advanced compiler, or optimizations to the translation process, could eliminate the temporary variable automatically.

Though all instance method calls are virtual method calls in Java [26, §7.7], the JIT can opti-

[1] In these experiments, no difference was detected between using `final` and a non-`final` subclass and/or method.

mize away most of the overhead in cases when inheritance is not used as shown in the entries for method invocations without inheritance. Once a true virtual function call is required, however, the dispatching strategy involving **switch** statements in the exploded object version is much faster than what the JIT is able to do, by nearly a factor of two, or more when no value is returned. Avoiding virtual function calls is likely to be a significant source of performance benefits in the exploded object version.

It may seem surprising that an explicit **switch** statement produced in the exploded object version is more efficient than what the compiler and JIT do with a virtual function call. As there are two different bytecode instructions that may be used to implement a `switch`, and their performance may vary depending on the number of cases in the `switch`, the microbenchmark was repeated with an additional twenty distinct subclasses. With the additional cases, which result in a very large **switch** statement in the translated version, exploded object operations involving a run-time type-check were slower by a factor of roughly two as compared to the microbenchmark without the extra subclasses. The running time of the Java version of the microbenchmark did not change. Even with the degradation, however, method dispatching in the exploded object version was still faster than in the Java version, though only by a factor of 1.4 instead of 2.6. Other versions of the Java virtual machine may yield different results based on the number of subclasses and the size of the resulting **switch** statement. Note that better analyses in the exploded object version can help minimize the number of cases in the **switch** statement by keeping track of which types might possibly have been assigned to a given reference and eliminating any types which could not have been assigned from the **switch**; the current implementation naively assumes the worst case and generates cases for all subclasses of the type being examined.

Results of the object creation microbenchmark are shown in Table 8.2. Results are shown for two different initial memory configurations, one in which no other allocation has taken place so memory is "clean," and the other after a number of creations and deletions have taken place so memory is potentially "dirty." Note that the presence of a generational garbage collector, as is found in the JVM used for these experiments, will ensure that allocation actually occurs in a "clean" memory environment even after repeated deletions. The difference in performance between the clean and dirty cases for the Java version is likely due to additional garbage collector costs in the dirty version.

Table 8.1: Microbenchmark results for instance field accesses and instance method invocations. The first column shows the operation, with (no inh) indicating no inheritance and (inh) indicating that inheritance was used. Times are in nanoseconds, and the performance factor column indicates the relative performance between the Java and exploded object version, with a value greater than one showing that the exploded object version was faster.

	Time (ns)		Performance Factor
Operation	Java	EO	
Field write (no inh)	3.0	5.0	0.60
Field write (inh)	2.9	13.0	0.22
Field read (no inh)	3.0	5.0	0.60
Field read (inh)	3.0	7.9	0.38
Method (no inh)	2.6	5.6	0.46
Method (inh)	23.7	9.0	2.63

Creating a new instance of an exploded object via `create` is more than four times as fast as creating an instance of a Java object using **new**. This shows that allocation-intensive applications, or portions of applications, should show considerable performance improvements in their exploded versions as compared to the original Java `Object`-based version.

Table 8.3 shows the cost of deleting an exploded object using the `destroy` method. Two different deletion patterns are shown, one in which a large number of deletions are performed on sequential objects, and the other in which the order in which deletions occur are randomized. Sequential deletion of exploded object instances is relatively quick, costing roughly the same as five Java virtual method calls (as measured in the first group of microbenchmarks, for instance methods where explicit subclass/superclass relationships are present). Random deletion is roughly twice as expensive, costing the same as eleven virtual method calls. The implementation of `destroy` used in the prototype implementation evaluated in these experiments performs no fewer than three instance method calls, including two to a Java collection (a `BitSet`) which in the implementation tested will in turn invoke up to three static methods, one method from a superclass, and six field accesses to get and clear bits used to track which slots are available for re-use.

Table 8.2: Microbenchmark results for object creation using **new** for Java objects versus `create` for the exploded object version. The first column shows the state of memory prior to allocation (clean and unused, or dirty and potentially fragmented due to past deletions), the second the average time per allocation of a Java object using **new**, the third the average time per allocation of the equivalent exploded object using `create`, and the rightmost column shows the ratio of the two times, with values greater than one indicating that the exploded object version was faster.

	Time (ns)		Performance Factor
State of Memory	Java	EO	
Clean	835	206	4.05
Dirty	900	216	4.17

Table 8.3: Microbenchmark results for exploded object destruction. Two deletion patterns are shown, one in which deletions occur sequentially and the other in which the order of deletions is randomized. Times are in nanoseconds.

Deletion Pattern	Time (ns)
Sequential	111
Random	254

8.3.2 Synthetic Benchmark Results

For the synthetic benchmark application, execution timing results were recorded for each of its three main phases as described in Section 8.2.1, namely database initialization (`initDB`), extracting the range of internal ID values (`PrintIDRange`), and gathering statistics about all `Person` objects in the simulation (`manipPer`). The total sum of execution time (`Total`) for these three parts was also recorded.

Three groups of experiments involving the synthetic benchmark application were performed. In the first, the set used to store all simulation objects for the `PrintIDRange` was not pre-allocated, and one million objects were present in the simulation. For the second, the set used by the `PrintIDRange` process was pre-allocated with the total number of simulation objects, and one million objects were present in the simulation. For the third, the `PrintIDRange` set was pre-allocated to hold all eight million simulation objects used. To pre-allocate set sizes, the total number of objects was passed to the set's constructor; depending on the exact implementation of the set being used (`java.util.HashSet` versus an array-backed specialized hash set implementation for use with exploded objects), specifying the initial set size may result in extra memory for the set being allocated.

Tests were run with varying initial heap sizes (using the `-Xms` and `-Xmx` parameters when invoking the Java runtime environment) to evaluate the effectiveness of the prototype implementation of exploded objects under varying memory conditions. An additional set of tests was run where the heap size was reduced to the minimum amount that allowed for completion of execution without errors. The running times of the Java object and exploded object versions were compared using the minimum heap size for each, to evaluate the effectiveness of the exploded object translation process in situations of minimum memory use; note that the actual heap size required for each version was different, and the heap sizes are indicated on the result graphs.

Figure 8.1 illustrates the performance factor results for experiments involving one million simulation objects, no pre-allocation of internal data structures, and executions under varying heap sizes. Figure 8.2 illustrates experiments under similar conditions, but where internal data structures (sets) were pre-allocated to a size sufficient to hold all simulation objects. Figure 8.3 shows results of experiments involving eight million simulation objects and pre-allocated internal data structures. For

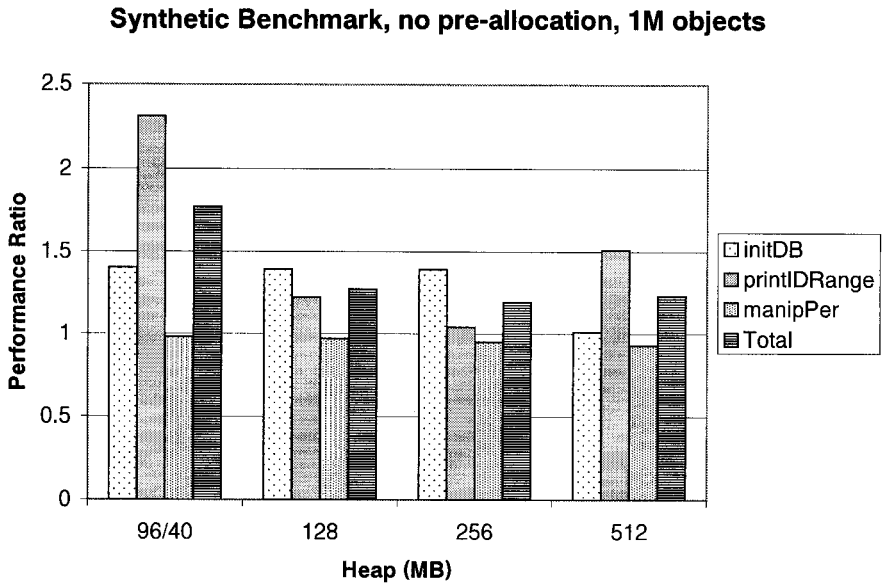

Figure 8.1: Performance figures for the synthetic benchmark application, using 1 million objects and run under different heap sizes. The 96/40 entry shows the results from a minimum-heap run, where the original Java object version required a heap of 96 MB and the fully-translated exploded object version required a heap of 40 MB to execute without running out of memory.

comparison purposes, two sets of results are displayed on the latter figure, with the 768E and 1024E runs on the right of the graph showing the performance gains resulting from manual elimination of all redundant uses of temporary variables in the code produced by the exploded object translation process.

The results in Figure 8.1 show that the benefits of the exploded object translation process as described in this work can be highly asymmetric. Database initialization, dominated by object creation, sees some benefits from exploded objects in most cases; this is not surprising given the relative costs of object creation as measured by the microbenchmarks. The `PrintIDRange` and `manipPer` passes show very different effects from the exploded object translation process. Both

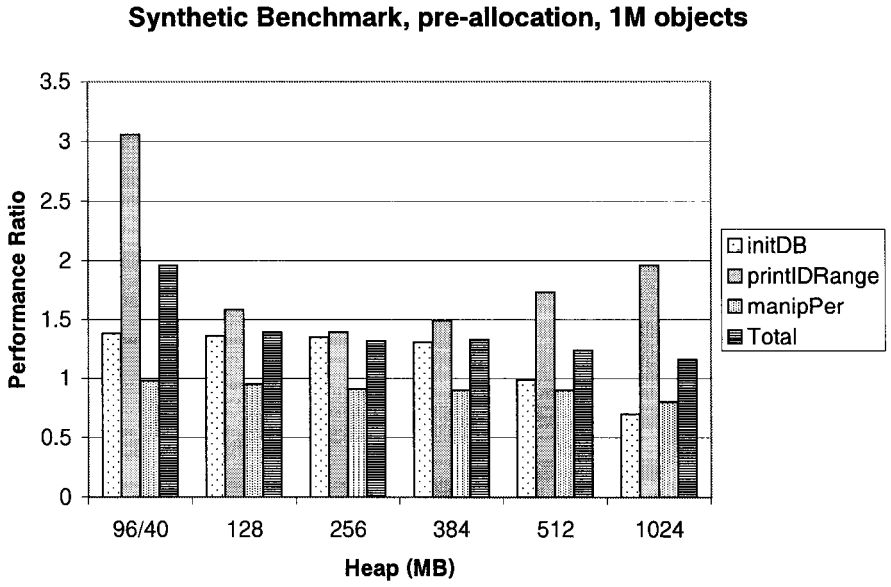

Figure 8.2: Performance figures for the synthetic benchmark application, using 1 million objects, pre-allocating collection sizes with that number, and run under different heap sizes. The $96/40$ entry shows the results from a minimum-heap run, where the original Java object version required a heap of 96 MB and the fully-translated exploded object version required a heap of 40 MB to execute without running out of memory.

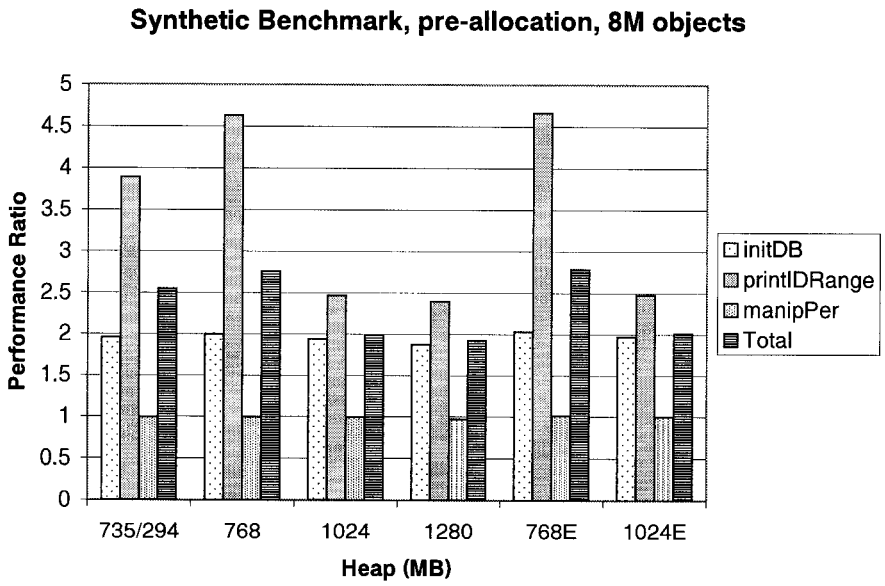

Figure 8.3: Performance figures for the synthetic benchmark application, using 8 million objects, pre-allocating collection sizes with that number, and run under different heap sizes. The 735/294 entry shows the results from a minimum-heap run, where the original Java object version required a heap of 735 MB and the fully-translated exploded object version required a heap of 294 MB to execute without running out of memory. The 768E and 1024E entries show the result of manually optimizing away all redundant uses of temporary variables in the code generated by the translation of the exploded object form; note that the performance gains from such optimization are minimal.

involve one dynamic dispatch to access data from an instance, iterating over sets of objects, but the code for each branch of the generated **switch** in the translation of `manipPer` is somewhat more elaborate. The `manipPer` translation involves several temporary fields, using a field from an exploded object instance as an array index as well as using part of a tuple containing index and run-time type information as a parameter. The difference in performance between the two parts of the experiment could be due to differences in generated code size between the two versions (e.g., one generates code that the JIT optimizes better, or that makes better use of internal processor code caches), performance issues with the unoptimized specialized data types presently found with the prototype implementation, different memory usage patterns that trigger garbage collection at different phases, or other factors.

The results in Figure 8.2 show that pre-allocating the internal data structures to hold enough objects provides a substantial performance boost to the `PrintIDRange` portion of the exploded object version in low-memory situations (the 96/40 minimum-heap case), but the relative advantage decreases quickly as more heap is available, until the total benefits are negligible at 512 MB of heap even while the gains in the specific `PrintIDRange` portion are measurable; the overall running time is not dominated by the `PrintIDRange` portion for simulations with one million objects.

When the number of objects in the synthetic benchmark application is increased to eight million, the results are somewhat different. As shown in Figure 8.3, improvements in running time in the areas that benefited from exploded objects in the previous version are more pronounced, and the overall results at the heap sizes tested show roughly a factor of two or more improvement for the exploded object version versus a factor of 1.25 to 1.5 for the version with one million objects. The 768E and 1024E entries show that only minimal performance benefits are from manually eliminating all redundant temporaries in the translated code resulting from the exploded object conversion process; this gain is not more than 2-4%.

8.3.3 UrbanSimlet Results

For the UrbanSimlet application, execution timing results were recorded for the total time taken for each of the three model components (the demographic transition model, residential mobility model, and the residential location choice model, as described in Section 8.2.2), as well as for initialization

of the object store and saving the results at the end of the simulation run. Additional timing results were recorded for the total elapsed time of running all models, time taken in extracting the set of applicable households from the object store, and the total execution time for each run (the sum of all model execution time, object store initialization, and saving the results at the end).

Each run involved one million households and one million grid cells, with ample available housing units to place all households. As for the synthetic benchmark application, heap size was varied using the `-Xms` and `-Xmx` parameters to evaluate application performance for the original Java object version and the fully-translated version using exploded objects under a range of memory conditions. For all runs, grid cell and household objects were translated as exploded objects. One set of runs involved tagging an additional object type with the **`exploded`** keyword, namely the `GridCell-RankPair` class used to store a `GridCell` location and a **`double`** representing its rank, used to store evaluations of different locations for each household during the residential location choice model. This process is very similar to a version of the residential location choice component from UrbanSim. Note that the `GridCellRankPair` class is not obviously suited for the exploded object translation process, as it contains only two fields, neither of which are small primitive types, and large numbers of instances are created and destroyed during the course of simulation execution. Experiments were run in which it was converted into exploded form to help judge the extent to which converting an apparently non-ideal object type into exploded form may affect performance.

Figure 8.4 shows the results from executing the UrbanSimlet with varying heap sizes, with the `GridCellRankPair` type being a normal Java Object. The 215/147 entry illustrates the result from running with the smallest possible heap for each version, where the original Java Object version of the UrbanSimlet application required 215 MB of heap to execute without running out of memory, and the translated version using exploded objects for households and grid cells required 147 MB. Figure 8.5 shows the results with the `GridCellRankPair` being converted to an exploded object for the exploded object-based version of the application. The 215/171 entry illustrates the result from running with the smallest possible heap for each version, with the original Java object version requiring 215 MB and the translated version using exploded object requiring 171 MB.

The results in Figure 8.4 show that, for nearly all heap sizes, all aspects of the UrbanSimlet see a boost in performance from the exploded object translation process as described in this work. The `End_Save` portion consistently shows the smallest performance gains, which is not surprising as

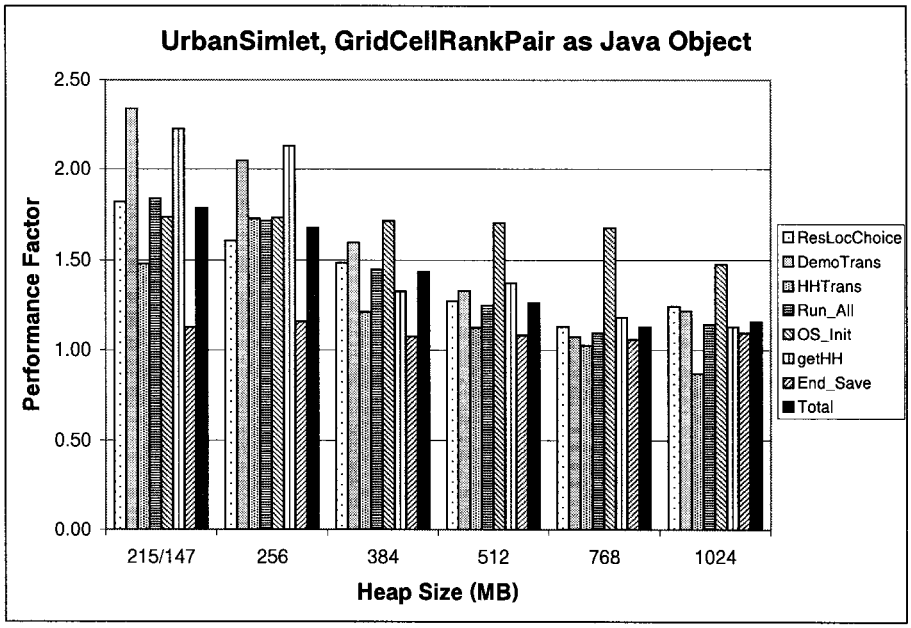

Figure 8.4: Performance figures for the UrbanSimlet application, using 1 million grid cell and household objects, run under different heap sizes. The intermediate objects used to store grid cell/rank pairs are represented as normal Java objects for these experiments. The 215/147 entry shows the result from a minimum-heap run, where the original version using only Java objects for all types required a heap of 215 MB and the fully-translated exploded object version, where grid cells and households were represented using exploded objects, required a heap of 147 MB to execute fully without running out of memory.

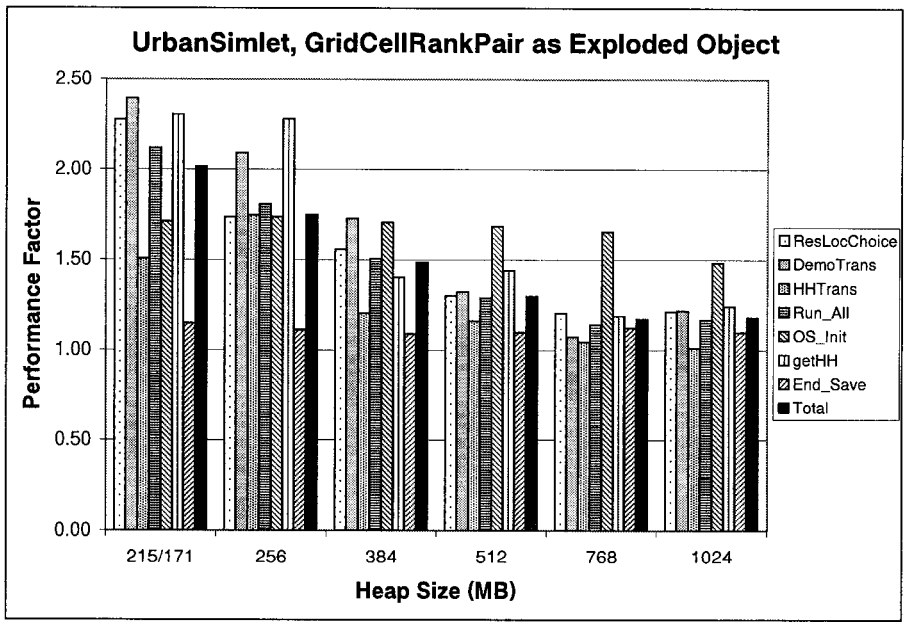

Figure 8.5: Performance figures for the UrbanSimlet application, using 1 million grid cell and household objects, run under different heap sizes. The intermediate objects used to store grid cell/rank pairs are represented as exploded objects for these experiments. The 215/171 entry shows the result from a minimum-heap run, where the original version using only Java objects for all types required a heap of 215 MB and the fully-translated exploded object version, where grid cells, households, and intermediate objects used to store grid cell/rank pairs were represented using exploded objects, required a heap of 171 MB to execute fully without running out of memory.

its running time is bound by disk writes and not object manipulation. Overall performance gains are modest, with the largest gain of a factor of 1.75 for the low-memory condition and falling to about 1.15 when a large amount of extra heap space is available. Initialization, which is bounded by memory allocation for the Java object version, shows consistent benefits from the exploded object translation process as the `OS_Init` result indicates.

The `getHH` result reflects total execution time required to iterate over all `Household` objects in the simulation, apply a filter to each one, and add ones that pass the filter into a set which is then returned. Performance gains at lower memory levels are likely due to slightly better handling of memory (hence less need for garbage collection) by the specialized data structures used to store the collection of `Household` objects to return, and the overall benefit drops off as memory becomes less of an issue at larger heap sizes.

The large performance gain in the `DemoTrans` model at lower heap sizes is likely due to the significant performance benefits from allocating additional exploded object instances for newly-created `Households` as compared to allocation in the Java object version, given that in the tests whose results are shown in the figure, total population (and total number of `Households`) increased during each simulation year.

Apart from the disk-bound `End_Save` portion, the `HHTrans` portion which shows the results of the household transition model that determines which households will try to relocate during a simulation year has the consistently smallest performance benefits from the exploded object translation process, dropping to a performance penalty when heap memory is plentiful. This model component involves a fair amount of non-object manipulation (looking up transition probabilities from tables), which can limit its potential gains from the exploded object translation process. It also involves a larger-than-normal quantity of introduced temporary variables in the translated version, which may contribute toward a decrease in performance.

Figure 8.5 shows the effects of converting the `GridCellRankPair` object type used to store pairs of grid cell locations and their desirability ranks by the residential location choice model. Even though large numbers of these objects are created and destroyed during every simulation year, and they do not contain smaller fields of primitive types, some performance benefits at lower memory levels (171, 256, and 384 MB) are seen for the `ResLocChoice` portion. As memory becomes less of an issue, overall performance drops to be roughly comparable with the version where the

GridCellRankPair object type was left as a Java object. Due to the unsuitability of the GridCellRankPair for conversion to an exploded object form, and possibly due to inefficiencies or flaws in memory usage of the specialized data structures used to store them, no memory savings are observed after the GridCellRankPair type has been converted to an exploded form. In fact, the latter version has a larger minimum heap size (171 MB versus 147 MB for the version where GridCellRankPair objects are normal Java objects), which, together with the minimal improvements in performance for most cases, show that choosing the wrong object types to convert to exploded objects can result in few to no benefits, and even hurt performance or boost memory requirements in some cases.

8.3.4 Data Synthesis Application Results

For the data synthesis application, execution timing results were recorded for each of its main phases as described in Section 8.2.3, namely creation of new household and grid cell objects (genHH and genGC respectively), placement of households into grid cells with available housing units (placeHH), accessibility computations involving radial neighborhood computations for each grid cell (computeAcc), saving household and grid cell objects to files (saveHH and saveGC respectively), and the total execution time (Total) not counting Java virtual machine startup or class loading time to the extent that class loading time is avoidable. As nearly all classes found in the application are referenced outside of any timed portions prior to execution, the overall impact of class loading should be minimal. Overhead due to just-in-time compilation is not explicitly measured, as it is amortized over the course of application execution.

For evaluation purposes, eight million household objects were created and placed onto a generated grid of grid cell objects measuring one thousand grid cells on a side, or one million grid cells. Experiments were run with varying heap sizes, as for the other two test applications. Figure 8.6 shows the results.

Several phases show very substantial benefits from the exploded object conversion. The genHH phase, in which household are generated based on supplied distribution of parameters, benefits from the generally large reduction in time required for allocation of an exploded object instance as compared to allocating a similar Java object. The placeHH phase, where a set of candidate grid

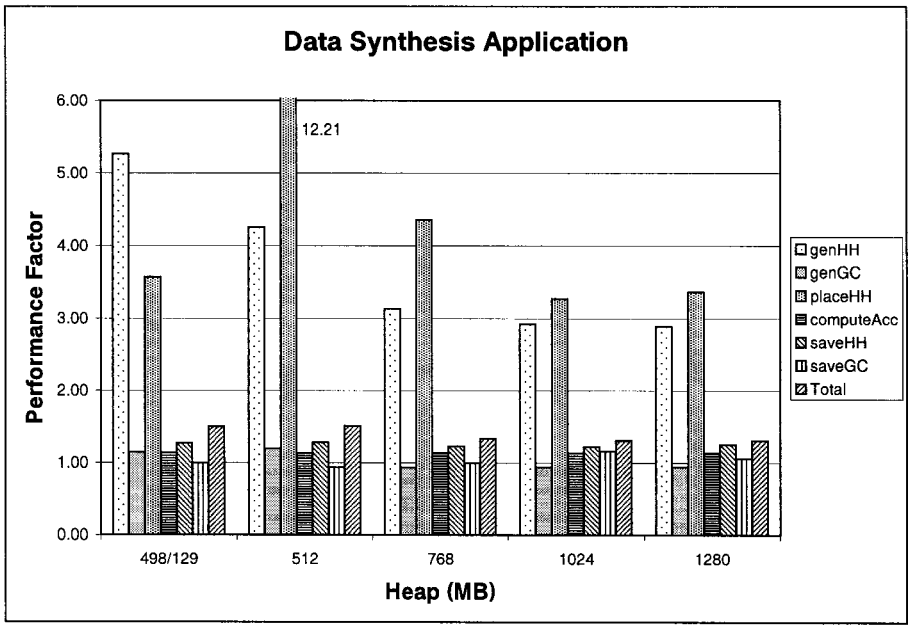

Figure 8.6: Performance figures for the data synthesis test application. Eight million household objects were created and placed into one million grid cells for each run. Household and grid cell objects were converted to exploded objects. The 498/129 entry shows the result from a minimum-heap run, where the original version using only Java objects required a heap of 498 MB to execute without running out of memory, and the translated exploded object version required 129 MB. The `placeHH` portion for the 512 MB heap run had an average performance factor of 12.2, which was truncated so as to make more visible the differences in the other phases.

cells are extracted (namely every one with at least one vacant housing unit) and households are placed into successive candidate grid cells as each grid cell is filled, shows a very large performance gain from the exploded object translation process. These gains may be due to differences in iteration implementation for canonical sets in the translated version, replacement of at least one object-level comparison with an arithmetic comparison in the translation version, the slight restructuring of `for`-loops caused by the translation process, improved opportunities for JIT optimization such as better register allocation caused by explicit temporaries or common subexpressions, or some combination of these and other factors. There is no single factor that is obviously responsible for the substantial gains, however, when one considers that any single factor causing the gains should cause similar gains in other portions of the test application or other applications with similar structures. A full explanation of the significant boost in performance would likely require analysis of the native code generated by the JIT and tracing its execution through an emulated processor, which is an extremely complex undertaking at best.

Generation of grid cell objects in the `genGC` phase does not show much of an improvement in the exploded object version, unlike other phases of other test applications involving object allocation. The generation of grid cell objects involves a substantial amount of table lookup based on random numbers which are used to populate the fields of a grid cell. This limits potential gains due to increased allocation speed in the translated version using exploded objects, and the translation process generates a rather large number of temporary variables in the innermost loops (twenty, as compared to one or two for most other inner loops) due to the structure of equations in the code. This results in a degradation of performance, possibly through an increase in code size due to assignment and manipulation of the multiple temporary variables, which in turn causes code cache misses in the processor, reduced JIT opportunities, or both; the end result is that the `genGC` phase has poorer performance in the translated version using exploded objects than in the original Java object version for most heap sizes tested. A good optimizing compiler should be able to eliminate many if not all temporary variables as introduced in code generation cases such as this one. Simulating the effect of an optimizing compiler by manually removing all temporary references in the `genGC` phase only resulted in performance improvement of $1 - 2\%$ over the exploded object version without the manual optimization, however, so the source of the performance degradation is not entirely clear.

Execution time of the data synthesis application is heavily dominated by the accessibility com-

putations reported in `computeAcc`. The accessibility computation involves taking the log of a weighted sum of employment and retail opportunities in neighboring grid cells, and is thus almost entirely bounded by non-object calculations.

8.4 Analysis of Results

There are several general results which can be drawn from the experiments described previously. These can be divided into three categories: general performance benefits of the exploded object translation process as described in this work, effects of low-memory situations on execution time, and memory savings resulting from the use of exploded objects.

8.4.1 General Performance Benefits

For most parts of the three test applications evaluated in this work, the exploded object translation process provides modest ($1.1\times$) to significant ($2.0\times$ or more) improvements in performance. The greatest benefits appear to be for portions of applications involving instantiation of exploded object instances, with certain forms of object manipulation showing improved performance. Improvements are likely due to a general reduction in amount of memory required for object representation, avoiding expensive virtual function calls to Java instances by replacing them with **`switch`** statements which can greatly improve performance as shown by the microbenchmarks, improved cache locality for data and/or machine code generated by the JIT at run-time, increased opportunities for just-in-time compiler optimizations such as former `Object`-level manipulations being replaced with primitive-type and array-access operations, and reduction or elimination of certain forms of overhead otherwise present for object manipulation.

8.4.2 Performance Under Low Memory Conditions

One advantage held by the exploded object translation process as described in this work is that its performance remains roughly constant as heap size is decreased, right up until the heap becomes too small for the application to complete execution. This is in marked contrast to the general behavior of Java applications, where decreasing heap size beyond a certain point results in substantial loss of performance. As the data presented in the previous graphs does not directly illustrate this effect,

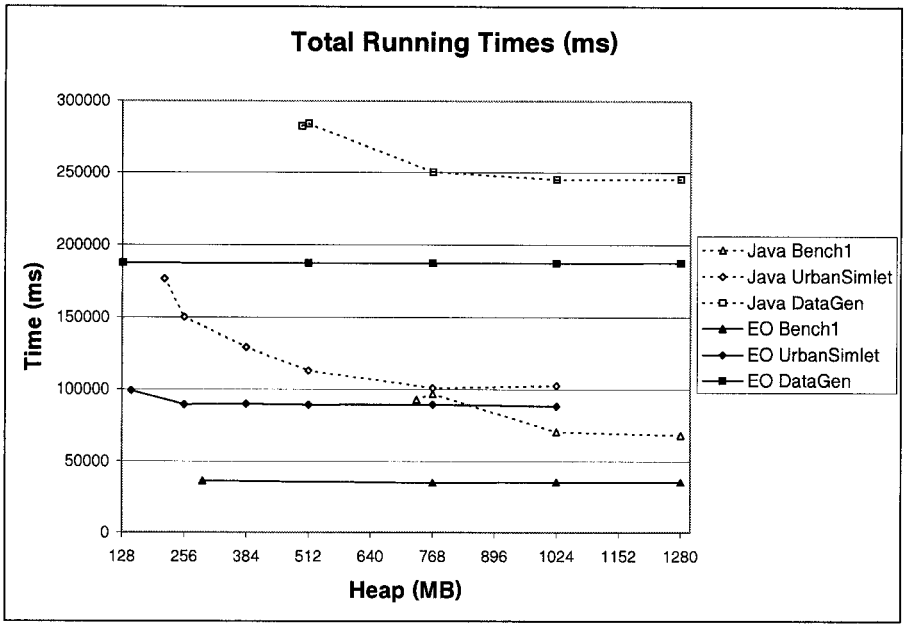

Figure 8.7: Raw total execution times for original Java object and translated exploded object versions of the three test applications. Examining the results from right to left, it is clear that the exploded object versions, marked with solid shapes and connected with solid lines, do not tend to degrade in performance as heap size is decreased, but rather tend to have a constant running time independent of heap size. The original Java object versions, represented by hollow shapes and connected with dashed lines, show that as heap size is decreased, running time tends to increase considerably.

Figure 8.7 shows the total running times for the various test application executions plotted against heap size. The set of three results for the exploded object versions, using the solid shapes for data points, show flat or nearly flat running time as heap size is decreased (looking at the results from right to left). The normal Java object versions show decreased performance as heap size is decreased, marked by increasing running times as show by the hollow-shaped data points connected with dashed lines.

Table 8.4: Comparison of memory usage between the original Java Object version of the test applications and the translated exploded object version. The memory reduction factor is obtained by dividing the minimum heap required by the original Java Object version with the minimum heap required by the exploded object version.

Test Application	Minimum Heap Size (MB)		Memory Reduction Factor
	Java Object	Exploded Object	
Synthetic Benchmark, 1M objects	96	40	2.40
Synthetic Benchmark, 8M objects	735	294	2.50
UrbanSimlet, `GridCellRankPair` as Java Object	215	147	1.46
UrbanSimlet, `GridCellRankPair` as exploded object	215	171	1.26
Data Synthesis	498	129	3.86

8.4.3 Memory Savings Resulting from Exploded Objects

Table 8.4 shows the reduction in memory usage resulting from the use of exploded objects. Reductions were measured by comparing the minimum heap size required for the original Java object version to execute without running out of memory to the minimum heap size required for the translated version using exploded objects.

Observe that in every case, the exploded object translation resulted in at least some reduction in memory requirements. Reductions were greater than a factor of two for a majority of the cases shown. These reductions in memory requirements, combined with the lack of performance degradation when running the exploded object translation of an application in low memory environments, argue very strongly that exploded objects can provide significant performance benefits. They also allow for modest (25%) to significant (4× or more) increases in the number of simulation objects without requiring more memory than the original Java object version, and available memory can be more completely filled with additional simulation objects without resulting in performance degra-

dation as compared with a Java object version of the same simulation. Using the exploded object translation approach described in this work, simulation developers may be able to choose between increasing the number of simulation entities while keeping the memory required by the simulation constant as compared to the original Java object version, improving performance while using less memory with the same number of simulation entities, or a combination of the two.

8.5 Adapting an Application for Exploded Objects

Part of each test application involved converting the original version that uses only Java objects into the version that makes use of specialized data structures, canonical sets for the exploded object types, etc. In practice, the conversion process was generally straightforward, though some of the prototype's rough edges made the process more complex than it otherwise may need to be. This section provides an evaluation of the conversion process involving the prototype implementation as outlined in Appendix B.

Identifying candidate objects for conversion to exploded object form was generally straightforward. For the test applications, it was clear as to which object types were simulation entities that satisfied the basic constraints for the exploded object translation process. One possible exception was the `GridCellRankPair` object type in the UrbanSimlet application; while it was present in significant numbers over the entire course of the simulation though not as much so during any particular simulation timestep, it did not contain multiple smaller data fields (just a **`double`** and an object reference to a `GridCell`), and large numbers of instances were created and destroyed in rapid succession during execution of the residential location choice model. Two different versions of the UrbanSimlet were evaluated for testing purposes, one in which the `GridCellRankPair` class was not converted into an exploded object type, and one in which it was.

Tagging candidate object types with the **`exploded`** keyword was a trivial edit. Due to the nature of the execution harness used with the prototype application, a copy of the code base was created upon which all modifications were performed. This allowed the original Java object version to be readily accessible at all times for purposes of comparisons, parallel testing, etc., but made extensions to the original version more cumbersome as the changes had to be reflected in every version of the code. For example, the collection of timing information was changed for the UrbanSimlet (to

measure total execution time for each model, instead of measuring total execution time for all models in each simulation year), and that required making identical edits in several copies of the code base.

Generating the first version of specialized data types was trivial, as it was handled by the harness.

Modifying the application code to make use of the specialized data types was trickier, as it required a fair amount of application-specific knowledge. For example, utility code that accepts a Java `Object` as a parameter may or may not be safe to manually convert to a version that takes a `Household`, for example, if the utility code may eventually add its parameter to a `java.util.HashSet` collection. If it is known that all use of the utility code can be divided into cases involving only exploded and non-exploded objects, then splitting the utility code into multiple specialized methods is safe. If a mistake is made, type mismatch errors will be generated at translation or compile time.

Additional complexity involved deciding which specialized data structures to use to replace existing general Java collections. For example, a `java.util.HashSet` may have been used because application semantics required object instance uniqueness and average-constant access time due to random accesses, or it may have been used arbitrarily when any Java collection would have sufficed. Picking the wrong specialized data structure can result in a severe performance impact (e.g., if all accesses to the collection are sequential, the overhead of a set may reduce performance if a list was more appropriate) and/or result in subtle violations of implicit application invariants (e.g., applications tracking object identity by their hash codes instead of references, or using `equals`-level equality versus ==-level equality, where a set-based collection will have different semantics than a list-based collection). A significant degree of application-specific knowledge is required to make the correct decisions with respect to offering performance improvements and preserving application semantics. To the extent that specialized data type templates have the same interfaces and semantics of Java data types[2], it is always "safe" to simply replace a general Java collection with its specialized equivalent, such as replacing a `java.util.HashSet` that is used to store `Household` instances with a specialized `HashSet_Household`, but doing so may eliminate possibilities for optimizations

[2]For instance, specialized `ArrayList` semantics are the same as for Java's `java.util.ArrayList`, but hash table semantics differ–the collection of keys or values obtained from a specialized `HashMap` is **not** backed by the keys or values in the specialized `HashMap`, unlike the results of invoking `keySet` or `valueSet` on a `java.util.HashMap`. Removing an entry from the set of keys from a specialized `HashMap` does not result in removing the corresponding value from the original specialized `HashMap`, unlike removing an entry from the set of keys obtained from a normal `java.util.HashMap`.

based on the use of other data structures.

Testing the modified application using intermediate generated versions of specialized data types was reasonably straightforward, in that none of the test applications had implicit or explicit reliance on any implementation-specific details of data structures. Testing consisted of running the modified version and comparing its output with the results from running the original Java object version. For other applications, testing might be more complex. For example, a common task in UrbanSim-style simulations is to grab a subset of all simulation entities of a given type, iterate over them in an arbitrary order, and perform some computation on each one, such as evaluating all households to determine which will attempt to find a new residence based on transition probabilities indexed by household characteristics. Differences in iteration order due to different implementation decisions of unordered collections can result in different simulation execution results, where both sets of results are "correct" but distinct. If repeatability of results is required for any portion of an application, care must be taken to ensure that all necessary implementation details or higher-level decisions are identical between all versions of the application. In practice, matching implementation details (e.g., choice of hash function for a hash set to ensure that iteration order is identical across implementations) is impossible, and instead a different data structure must be chosen at the application level (e.g., an ordered set instead of an unordered one) with a matching selection made during the conversion to the exploded object form.

Generating the fully-translated version of the application involved executing the harness with different command-line parameters, as well as some minor editing of `import` information to ensure that any Java types which might have been moved from one package to another as a result of the generation process would still be found during class loading. In one case, this process was not immediately obvious; due to the selection of target packages for generated code made during the translation of the UrbanSimlet, the non-exploded version of the `GridCellRankPair` class referred to the exploded version of the `GridCell` type, which was no longer located in the same package and thus needed an explicit `import`.

Testing the fully-translated version of the application involves compiling and running the translated code and comparing the results with the results from the original. Due to changes in implementation details of certain collections as compared to the original Java collections, any implicit reliance on iteration order through unordered collections will likely result in the translated version

producing different output that is still correct; this is the same issue found during testing of the modified application using intermediate forms of specialized data types, but is much more likely to cause differences in practice with the prototype implementation. The templates for intermediate forms of specialized data types are simple wrappers around the original Java collection type, such as `HashSet_Household` wrapping a `java.util.HashSet`, whereas the templates for the final versions of specialized data types are hand-written versions of those data types which will almost certainly involve different decisions on hash functions and growth/re-hashing strategies, for example.

Overall, the conversion process to modify the original Java object versions of the test applications into the versions using exploded objects was not a difficult task, but it did require more application-specific knowledge than was originally expected. Adding analyses to the prototype implementation could help reduce the degree of application-specific knowledge required for the conversion process, but some domain expertise and familiarity with the application will likely always be required.

Chapter 9

RELATED WORK

There are several fields of work which are related, more or less directly to the general exploded object translation problem. These include object inlining, object layout optimizations, header word compression, and data size optimizations.

9.1 Keunwoo Lee's Qualifying Exam

After several discussions with the author regarding the initial application of exploded objects to UrbanSim, Keunwoo Lee's Qualifying Exam paper [24] presents a systematic approach to exploding objects that preserves all Java object semantics. His proposed approach has several key features that differentiate it from the Restriction Approach presented in this work, namely programmer-controlled object layout, full support for synchronization and reflection involving individual exploded object instances, and support for garbage collection of exploded object instances. Unlike the Restriction Approach, Lee's proposal requires a customized Java virtual machine.

In Lee's proposal, programmers explicitly annotate object definitions to partition fields into collections called shards, and then provide further annotations indicating how shards are to be allocated into arenas. At instantiation time, the user explicitly chooses one of the possible layouts, which allows for a great deal of flexibility in selecting optimized layouts for specific object and field usage patterns.

The run-time environment in Lee's proposal provides support for the full range of reflection involving exploded objects, unlike the Restriction Approach which wholly disallows the use of reflection with exploded objects. Lee's proposal also allows for synchronization on individual exploded object instances, as well as full support for inheritance, including casts to interfaces, and implementation of the same interface by both exploded and non-exploded type. The run-time environment supports garbage collection of exploded object instances as well, effectively allowing

exploded objects to have the same semantics as normal Java objects but providing possibilities for layout-related optimizations. In this sense, Lee's proposed approach to exploded objects is much more general and flexible than the Restriction Approach.

However, unlike the Restriction Approach, Lee's work requires a customized Java virtual machine to support its features. The flexibility of object layout options requires virtual machine support to efficiently handle different options regarding header word optimizations. Additional virtual machine customization is required to support garbage collection. The requirement of a customized virtual machine is in direct conflict with one of the requirements of the Restriction Approach, rendering Lee's proposal unsuitable as a basis for this work.

Lastly, Lee's work consists of a proposal only, without any implementation to provide empirical evaluation of its effectiveness. The flexibility of his approach may make efficient implementation difficult, though it may be possible to impose restrictions on his work to allow for more efficiency, much as the Restriction Approach has done.

9.2 Object Inlining

Object inlining [12, 43] is a term used to refer to a class of optimizations in which a reference to an object is replaced with in-line fields, such as replacing a reference to object representing a pair of integers with two local `int` variables. More sophisticated forms of object inlining handle inlining objects passed to procedures, invoking methods on an inlined object, etc. Inlining an object generally improves performance by eliminating indirection for field accesses, allowing more data to be stack-allocated instead of heap-allocated, and can facilitate other types of optimizations.

From one perspective, exploded objects are an extreme form of global object inlining, so there is naturally some overlap between the two areas. However, as object inlining is generally more of a local optimization than a global optimization, it is largely complementary to exploded objects. Most of the same techniques that can be applied to inline a "normal" object can be applied to inline an exploded object, though inlining an exploded object would effectively "un-explode" it, and may not be generally applicable to the typical usage patterns of exploded objects. Object inlining could still be of benefit when applied to non-exploded objects in an application making use of exploded objects, however.

Some specific applications of object inlining as related to this work are described below.

9.2.1 Automatic Inline Allocation

Dolby [12] describes a process for automatic inline allocation of objects that handles inlining of objects within containers and other complex pointer relationships. Analyses determine when objects that would normally be heap-allocated can be replaced with a collection of stack-allocated fields and specialized methods. In addition, the process allows for "collapsing" inner objects into a flatter representation to avoid additional layers of indirection, when alias analysis indicates collapsing is safe.

The basic analyses and transformations found in this work are very similar to those in Dolby's paper. The added complexity of handling nested objects is generally not required for UrbanSim-style simulations, in that objects are rarely nested. Simulation objects may contain references to other objects, such as a reference to the `Household` of which a `Person` is part, but the referenced objects have existence outside of the referrer. Nesting or container relations do not generally exist between simulation entity objects in an UrbanSim-style simulation.

Dolby's work performs cloning and specialization to handle inlining of objects whose types may vary at run-time. This is different from the approach used in this work, in that this work explicitly tracks run-time type information and handles it in a uniform fashion instead of generating specialized code for all possible combinations.

9.2.2 Zoran Budimlić's thesis

In his 2001 thesis [6], Budimlić describes several compiler optimizations applied to Java, including object inlining and method specialization. His work allows for inlining of three types of Java objects: local objects, global objects, and arrays of objects. Of these, the most relevant to this work is the handling of arrays of objects.

The system described by Budimlić converts arrays of objects into arrays of the fields contained by the objects, and the translation is nearly identical to that proposed in this work, as are the translations for method specialization. In addition, many of the analyses used to determine when it is safe to inline an object, or array of objects, are nearly the same as the analyses used to determine when it

is legal to represent an exploded object reference in its efficient form versus when a more expensive wrapper or other conversion is required.

The work on exploded objects as described in this work differs from Budimlić's work in three primary ways: usage of semantic information, support for run-time type information, and explicit support for collections. Budimlić's work does not attempt to extract, or use, any semantic information from the program about the intended use or nature of objects before attempting to perform object inlining, so that all object types are subjected to the same degree of analysis. Exploded objects are explicitly tagged by the programmer to show that specialized analyses and transformations should be attempted, and this tagging information allows for more specialized analyses involving a more limited scope. In particular, alternate data structures and optimizations may be used with exploded objects that may not be suitable for general Java objects. As a result, Budimlić's approach cannot identify cases where global transformations based on an object's usage patterns would be advantageous, whereas tagging an object type as exploded explicitly marks that type as being suitable for the global exploded object transformation and indicates that other optimizations may be fruitful due to the global usage patterns of the type.

Budimlić's work does not provide for run-time type information; if sufficient static type information cannot be extracted by the analyses as described in Budimlić's work, inlining does not occur. Exploded objects provide explicit support for run-time type information, which enables certain forms of optimization and transformation even when static type information is lacking.

Thirdly, Budimlić's work does not perform inlining or other transformations of "higher-level" object-oriented structures such as sets, maps, etc. While some usage of objects associated with such collections may be inlined, such as an object returned by an `iterator` over a collection, the internal representation of the collection or the objects within it may not be subject to further optimization. Exploded objects as described herein provide explicit support for transformations involving collections of objects, including specialized representations, iteration, etc., beyond what Budimlić's transformations allow.

For the most part, object inlining optimizations as described in Budimlić's work are complementary to exploded object transformations. The same localized transformations that are applied to inline normal Java objects in Budimlić's work can be applied to further optimize an exploded object by replacing references to exploded-object arrays by inlined local variables, for example.

Object inlining can further complement the exploded object approach by improving performance of specialized data structures generated as a result of the exploded object translation process, such as iterators within specialized sets.

9.3 Object Layout Optimizations

Object layout optimizations encompass the broad class of optimizations involving altering how objects and their fields are organized in memory in order to improve performance. These optimizations are largely orthogonal to exploded objects, but merit further discussion.

9.3.1 Cache-Conscious Data Layout

One object layout optimization involves altering how object fields are allocated or organized in memory so that commonly-used fields fit more readily into processor caches. Chilimbi et al. [7] describe techniques for reorganizing object field layout in C by using clustering to increase the likelihood that commonly-used fields fit within a single cache block. They also alter object allocation patterns to make better use of cache locality, such as allocating sequential elements in a linked list sequentially in memory. Their methods rely on static analysis to determine clustering, and dynamic analysis through profile data to tune allocation patterns.

It is unclear to what extent this form of low-level optimization would be effective for the Java language, in that each implementation of the Java virtual machine may or may not map closely to the underlying hardware which includes generally small processor caches. However, the use of just-in-time compilation and other techniques to produce partially or wholly native code from Java should profit from this style of optimization. As exploded objects have already been broken apart into parallel arrays, there may not be much opportunity for performing cache-conscious storage transformations. However, the analyses used to alter allocation patterns could potentially be combined with exploded objects to allow for tuning of exploded object allocation within each collection of parallel arrays, for example by altering the ordering in which array slots are filled by creating exploded object instances.

9.3.2 Data Layout Optimizations to Enhance Spatial Locality

One class of data layout optimizations is designed to enhance spatial locality of data, so that groups of data that are commonly used together are arranged together in memory. Kandemir et al. [22] describe a process for transforming data layouts to improve performance on uni- or multiprocessor systems.

This style of optimization is generally orthogonal to exploding objects. However, it could be applied to optimize the layout of the collections of parallel arrays generated by the object exploding process, as well as to optimize other arrays contained within a simulation.

9.4 Header Word Compression or Elimination

A range of techniques has been applied to allow for compression of object header words in Java and other languages, or to allow for outright elimination of some or all header words.

Bacon et al. [4] describe a range of methods for eliminating many of the normal header words from Java objects while preserving full Java object semantics. Techniques include "stealing" bits from the type information block by using the high-order bits of a header word as an index that references a class pointer, or replacing always-set or always-clear bits in the TIB with bits used for other purposes, replacing the per-object lock with a short pointer to an entry in a lock nursery (collection of locks), eliminating the default hash code or replacing it with a simple function based solely on the object's address, and/or collapsing more general-purpose header words into smaller collections of bits through compression or packing. Their work demonstrates that memory savings can be significant, on the order of 21% with respect to normal object headers, with performance generally improving by a similar amount except when large numbers of virtual functions are invoked. Their approach has the advantage of preserving full Java object semantics and requires no modifications to the source code by the programmer.

As the implementation of exploded objects as described in this work eliminates all header words save minimal run-time type information as required, the above approaches to reducing or eliminating header words are not directly relevant. However, they would become useful if exploded object semantics were modified to more closely match those of normal Java objects; for example, the lock nursery concept from [4] would be useful if synchronization were to be allowed on instances of

exploded objects.

9.5 Data Size Optimization

A range of techniques to reduce data size are discussed by Ananian et al. [1], including elimination of object fields with constant values, size reduction of primitive types based on the range of values used, elimination of fields with common default values or usage patterns, compression of the class pointer, and rearranging object fields to increase the opportunities for byte-packing. Experimental results demonstrate a significant reduction in heap requirements, with a generally positive impact on overall execution time.

Several of these reduction techniques could be applied to the implementation of exploded objects as described in this work. Eliminating object fields with constant values and reducing the sizes of primitive types based on actual data ranges would both be directly applicable to exploded objects. The special cases of field elimination for hash code and lock storage discussed in the paper are not relevant to exploded objects, as the exploded object representation already eliminates storage of a precomputed hash code value, and exploded objects cannot be used for synchronization so no per-object lock need ever be stored.

Class pointer compression, which is achieved in Ananian et al.'s work by replacing the normal Java object class pointer with a smaller offset into a separate table, is roughly analogous to the use of the smallest possible data type for storing run-time type information in the prototype implementation of exploded objects described in this work, with the prototype implementation also performing the equivalent of inlining the table lookup and resulting dispatches.

Byte packing of fields to help reduce memory wasted due to data alignment issues is not relevant to exploded objects, as storing fields in parallel arrays wholly eliminates the per-object overhead that would be incurred due to data alignment.

Chapter 10

FUTURE RESEARCH DIRECTIONS

There are a number of ways in which the prototype implementation could be extended or enhanced in the future. The underlying translation scheme as described in this work could be modified in a number of ways as described below. There are also ways in which the restrictions imposed upon the characterization and usage of exploded object could be relaxed, and the overall exploded object approach to object layout optimizations could be applied to other languages or in other fashions. The impact of future advances in Java stand-alone and just-in-time compiler technology is considered as well.

10.1 Modifications to the Prototype Implementation

The prototype implementation could be enhanced in several ways, without substantially changing its structure or the underlying translation scheme or treatment of exploded objects. These modifications can be divided into three categories: enhancing the quality of code generated during the translation process, enhancing the performance of the specialized data types as provided in template form with the prototype implementation, and making the prototype implementation easier to use.

10.1.1 Enhancing Generated Code Quality

The code generated by the translation process in the prototype implementation could be enhanced in several ways. The current translation process can result in the generation of redundant temporary variables and accompanying redundant assignments, which negatively impacts performance. The performance penalties likely arise from increases in code size which impact code cache performance, or through redundant assignments that are not eliminated by the compiler. In some cases, additional static analysis would be able to determine whether a dynamically-dispatched call involving an exploded object type can be replaced by a direct call instead of a generated `switch` block

with only a single **case** besides the default error case.

Method inlining [2, 36] could be used to eliminate the per-invocation overhead which may be higher for methods in exploded objects than for methods in normal Java objects, due to the necessity to pass at least one extra parameter to the method (the exploded object's index in its canonical set) and possibly another level of field indirection to access the method code through the static reference to the dynamic instance of the canonical set versus a local reference access. Adding the dispatch through the field reference is likely the more significant of the two costs, as the dispatch results in a virtual function call.

Object inlining [6, 12, 43] is an optimization technique that is generally compatible with the exploded object translation process as described in this work. Applying it to inline exploded object references could improve performance by avoiding at least one level of field and array access indirection to access an exploded object's field, and in some cases, to eliminate the generation of any form of run-time checks that are required to ensure that accessing a shadowed field in a subclass reads or writes the appropriate version of that field. Inlining an exploded object instance would likely only be of benefit when a small number of instances are used repeatedly, or the increase in code size resulting from large numbers of temporary variable initializations may overwhelm potential savings from the inlining process. As repeated use of small numbers of instances of simulation entities (e.g., `Households`) does not happen very often in UrbanSim-style simulations, object inlining may be more effective when applied to non-exploded types.

Note that continuing advances in just-in-time compilation techniques may obviate some of these optimizations at the exploded object translation or code-generation level, as the JIT may be able to do a more thorough job.

10.1.2 Enhancing Specialized Data Type Performance

The templates provided with the prototype implementation that are used to generate specialized data types are untuned and unoptimized. In particular, for the current hash set and hash table templates, no particular effort has been made to ensure that the hash functions involved are well-suited to typical application data, that hash table sizes are increased at a reasonable rate with respect to balancing frequency of table re-sizing versus minimizing overhead from excessive table size or growth-related

copying, or to re-hash data under certain conditions to help minimize the number of probes required during table access. It is well known [9, Chapter 12] that poor choices of hash function, collision resolution, and table growth and re-hashing strategies can negatively impact performance of hash-based data structures. As tuning data structures is a generally well-studied problem, can be highly application-specific, and is not germane to the exploded object translation problem as described in this work, no special efforts have been undertaken to address the tuning issue for the prototype implementation. Tuning of specialized data structures can only improve the performance of exploded objects in the future.

Some application domains may be willing to sacrifice memory for additional performance, or have even more stringent memory requirements than is typical. For either case, the specialized data type templates could be modified or replaced to reflect domain-specific criteria. The currently-supplied templates are designed for a general case, not tuned for either of these alternate environments.

10.1.3 Increasing Usability of Prototype Implementation

The present prototype implementation is somewhat cumbersome to use, as described in Appendix B. The prototype implementation could be modified, at a non-trivial increase in complexity, so that template instantiation and other code generation involves programmatic generation of Polyglot abstract syntax trees instead of writing files which are later processed by Polyglot. This process could potentially eliminate the intermediate generation steps to whatever extent desired; one intermediate generation step will likely still be of use to allow for testing the modified application before running the translation process. Internalizing code generation could improve the prototype's ability to detect errors that would otherwise be left for the Java compiler to catch, as well.

The prototype implementation makes some effort to provide useful error messages, but more and better error messages could be provided. In particular, errors that are introduced during the code-generation process (such as compiler bugs) are not caught by the prototype implementation proper, but rather by the Java compiler used to compile the generated code. This results in error messages that refer to the translated code, which is not always easy to understand due to the restructuring that takes place during the translation process. It can be particularly difficult to map back from trans-

lated Java code to the original version of the code involving exploded objects in some cases, such as when non-local transformations of the original were introduced to handle run-time dispatching and/or the introduction of temporary variables of exploded object types requiring run-time type information. Addressing this "correspondence problem" between errors in the Java code generated by the translation process and the original exploded object-using code would be non-trivial, but potentially helpful for users of the prototype implementation.

A more radical modification that would combine aspects of both of the above usability enhancements would be to integrate the prototype implementation into an integrated development environment such as Eclipse [34]. Doing so could make the entire process of converting an application to make use of exploded objects much easier as compared to the current process involving the prototype implementation. In particular, the IDE could transparently hide the intermediate code generation steps, assist with or automate much of the currently manual conversion process, and help match errors in the translated code with their sources in the original exploded object-using version of the code.

10.2 Modifications to the Translation Scheme

There are several ways in which the translation scheme could be modified for purposes of potential enhancement or experimentation, namely alternative representations for references to exploded objects, or alternative representations for instances of exploded objects.

10.2.1 Alternative Reference Representations

The implementation of exploded objects as described in this work uses an `int` for a reference to an exploded object, accompanied by a `byte` for run-time type information when required. For some domains with very small numbers of exploded object types, it might be feasible to use a bit-stealing approach to represent an exploded object reference with run-time type information in a single `int` by using high-order bits to store run-time type information. On platforms with 64-bit words, a single word (`long`) should provide enough space for this approach. Alternatively, a Java 64-bit `long` could be used to represent a reference plus run-time type information in a uniform fashion for machines with varying word sizes, but this approach may be considerably less efficient on 32-bit architectures,

and requires more memory to store large numbers of references as compared to a bit-stealing or `int` plus `byte` approach.

10.2.2 Alternative Internal Instance Representations

There are several other internal representations for exploded object instances that may be worth investigating for future versions of the prototype implementation.

The current prototype implementation uses an agglomeration approach to handle inheritance. Each instance of an exploded object type contains versions of all fields from all superclasses. An alternative approach would be to allocate distinct instances of each superclass and connect them with forwarding pointers. This approach might boost performance in some circumstances when only fields defined in a superclass are referenced for large numbers of objects by increasing cache locality and/or the effectiveness of any pre-fetching, but would require additional space to store forwarding pointers. This approach would also avoid requiring dispatching for field accesses, which would boost performance.

Another approach would be to make run-time type information implicit instead of explicit by breaking the internal arrays in the canonical sets into sections, so that the index would encode the run-time type in a different way than the bit-stealing approach described previously. Checking the run-time type of an exploded object instance would be more expensive than it is at present, as it would require some arithmetic, but it could save memory by potentially avoiding any overhead for run-time type information. For example, all instances with indices i in the range $[0..n]$ could have one run-time type and be stored in array index i in one section, instances with indices in the range $[n+1..2n]$ could have another run-time type and be stored in index $i-n$ in another section, etc. This approach would only be appropriate in cases where the number of exploded object types and instances were sufficiently small that a bit-stealing approach would likely work as well, though with somewhat finer granularity. For example, a bit-stealing approach would require three bits to store run-time type information in an environment with six different types of exploded objects, reducing the maximum number of exploded object instances by a factor of eight, but a sectioned-array approach would require only a factor of six reduction in the maximum number of exploded object instances at most. For both approaches, the bits used to store run-time type information would

be "stolen" from an already-required field, such as the high-order bits of the index that represents a reference, thereby avoiding any separate storage space for run-time type information.

10.3 Relaxing Restrictions

It may be possible to relax a number of restrictions placed on the usage of exploded objects and on the mixing of exploded and non-exploded object references. Some possible restriction modifications are described below.

10.3.1 Adding Support for Interface Implementation

The current prototype implementation has a restriction that no interfaces may be implemented by exploded object types. This restriction could be relaxed in several ways. One approach would be to allow for "exploded" interfaces, where certain interface types would be tagged as **`exploded`** and could be implemented by exploded object types; non-exploded object types would not be able to implement exploded interfaces. If exploded types could not be cast to exploded interface types, adding support for exploded interfaces would be relatively straightforward, requiring only a local analysis to ensure that each exploded object type implements all methods defined by all exploded interfaces it implements. If exploded types could be cast to exploded interfaces, run-time type information would be required to appropriately handle dispatching using exploded interfaces.

Another approach to relax the restrictions on interface implementation would be to allow exploded objects to implement any kind of interface, and be cast to any interface type they implement. This option would require efficient handling of exploded objects being converted to Java `Object` subclasses, as an interface can be cast to `Object`. Modifications to the Java virtual machine may be required to efficiently handle this form of subsumption.

10.3.2 Supporting Inner Classes

The present version of the prototype implementation does not support exploded object types with inner classes, or exploded inner classes of non-exploded object types. Future versions of the prototype implementation could be modified to remove either of these restrictions in several ways. An exploded object type with an inner class could be converted into a pair of exploded object types,

with accessors from the inner class being modified to reference the outer class, providing the appropriate privacy modifications are made. Relaxing access restrictions to fields of exploded objects in the translated code should generally be safe, as all accesses to fields of exploded objects could be type-checked against the original access mechanisms independently of the ones resulting from the translation process.

Converting an inner class of a non-exploded object type into an exploded object type would be less straightforward, as it may not be as feasible to make the same form of access restriction modifications to a normal Java object if it is used in contexts other than in conjunction with exploded objects. While the access restrictions are relaxed to compile normal Java inner classes, access restrictions would need to be relaxed in a similar fashion for the surrounding non-exploded class to ensure that the exploded inner class (as contained in a canonical set) could still access its originally-surrounding class in the same fashion as before the exploded object transformation. Relaxing this restriction may be of less importance, as in practice there are rarely large numbers of instances of an inner class without there being large numbers of the corresponding outer class such as a one-to-one mapping, or where there is no direct connection between the inner class and the outer class such as for helper classes such as tuples declared as inner classes as a convenience rather than out of necessity.

Adding support for static inner classes would be more straightforward than either of the above cases, as handling them would at worst be a simple matter of extracting them from their surrounding class and placing them in a separate file for separate compilation, and updating any references to reflect the movement.

10.3.3 Supporting Garbage Collection of Exploded Objects

One of the most significant differences between normal Java objects and exploded objects as described in this work is that exploded object instances are not subjected to the same garbage collection procedure that normal Java objects are. Adding support for garbage collection of exploded objects would require substantial modifications to the run-time environment. While it might be possible to implement exploded object garbage collection at the user level, doing so would be unlikely to be efficient enough to be fruitful. Instead, modifications to the virtual machine would be needed,

which would make exploded objects less attractive for general use as the translation system would be tied to a specific customized virtual machine.

10.4 Other Applications

In addition to the above extensions and modifications, there are several other directions in which exploded objects could be taken. The two most noteworthy of these are the application of exploded objects to other object-oriented languages, and automated detection of candidate object types for transformation into exploded objects.

10.4.1 Applying Exploded Objects to Other Languages

The general approach to exploded objects as described in this work should be at least somewhat applicable to other object-oriented programming languages when large numbers of object instances are present in an application. It is difficult to predict exactly what performance and/or memory reduction benefits exploded objects may bring to any particular language, but in general, object-oriented languages with larger per-instance overhead (e.g., many header words) and opportunities for efficient representation of primitive data types (e.g., where not all object fields are always wrapped objects) have the potential for performance benefits from the application of exploded objects.

The allocation/deallocation management semantics of exploded objects may be more palatable in other languages as compared to Java. The current requirements for explicit deallocation of exploded object instances would match well with languages that do not have garbage collection, such as C++. Languages that provide direct support for customization of object allocation, such as overriding **new** in C++ or Common Lisp's meta-object protocol [42] would be able to more cleanly incorporate the use of canonical sets to store all instances of exploded objects, as compared to the use of **new** for Java objects and `create` method invocation for exploded objects as required in the current prototype. This assumes that the compiler itself does not provide for translation of **new** into a call to `create`, as the prototype implementation provides. For the specific case of C++, there may not be much reason to apply exploded objects given that the per-instance overhead in C++ is generally very low (e.g., just a virtual function table pointer, no pre-computed hash code or synchronization lock). For certain access patterns, such as simple operations applied to a small subset of

fields of a very large number of object instances, representing fields of object instances in parallel arrays rather than in separate per-instance blocks may still provide benefits.

10.4.2 Automatic Identification of Exploded Object Candidates

One way to improve the usefulness of the exploded object translation approach as described in this work would be to provide some form of automatic mechanism for identifying types as candidates for the exploded object transformation process. A combination of static and dynamic (profile-driven) analysis could be used to identify types that are present in large numbers during a simulation, are not used for synchronization, and contain significant numbers of smaller fields, and then recommend that those types be considered for the exploded object conversion process. Depending on the sophistication of the dynamic analyses, it may be feasible to have the suggestions include hints on tuned versions of specialized data structures to use with each given type of exploded object, or even within different contexts where collections of exploded object references are manipulated.

10.5 Future Applicability of Exploded Objects

As just-in-time compiler technology continues to improve, the overall benefits from the exploded object translation process as described in this work may be lessened. For example, better JIT technology may allow for automatic reduction in the size of object instance headers, provide for more efficient packing of small data fields, or provide better data locality for certain access patterns than the exploded object translation process provides. In the limit, a JIT or a stand-alone compiler could perform similar radical reorganization of internal object representations in a fashion analogous to the exploded object translation process described herein, and thereby gain equal performance benefits and reduction in memory requirements. However, it is likely that code generated as a result of the exploded object translation process will benefit equally from general improvements in JIT and other compiler technology advances; the translation process described in this work generates pure Java code as output precisely to leverage future advances in Java stand-alone and just-in-time compiler technology.

Chapter 11

CONCLUSIONS

This dissertation presents the technique of object exploding, a novel approach to object representation in the Java programming language. Through experimental results involving a prototype implementation of the translation process, it is shown that the exploded object translation process can result in substantial reductions in memory requirements and provide significant increases in performance for certain patterns of object usage common to a broad class of simulation and other applications.

In situations where memory is limited, either by the nature of the system or by the desire to increase the number of objects in a simulation, the exploded object translation process described herein provides substantial benefits. Unlike applications involving only Java objects, applications that have been translated to use exploded objects show no degradation in performance as available memory becomes more and more limited.

The major contributions of this dissertation are threefold: an examination of the space of potential approaches to the exploded object translation optimization scheme, a complete description of a particular translation scheme along with justifications for the various design decisions, and a prototype implementation used to generate experimental results showing that the particular translation scheme described in this work can provide substantial benefits in terms of increased performance and decreased memory requirements.

When considering the general problem of converting Java objects into an alternate form based on parallel arrays in order to improve performance, there are many different tradeoffs that can be made in terms of preserving Java object semantics, support for various Java language features, usage restrictions, etc. This dissertation defines the space of alternatives for these and other choices.

One specific approach to the exploded object translation, namely the restriction approach, is defined in this dissertation, along with motivations for its applicability to a significant range of simulation applications, the ramifications of the choices made by the restriction approach, and a

detailed translation strategy by which the restriction approach can be realized.

Lastly, a prototype implementation of the restriction approach to the exploded object translation problem is described, along with a number of experimental results involving three different test applications. Experimental results using the prototype implementation show that the restriction approach can provide significant benefits in terms of increased performance and reduced memory requirements. These experiments serve to support the computer science hypothesis that the exploded object approach to Java object representation can provide factor-of-two increases in performance and factor-of-two reductions in memory usage.

BIBLIOGRAPHY

[1] C. Scott Ananian and Martin Rinard. Data size optimizations for Java programs. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, June 2003.

[2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 134–145. ACM Press, 1997.

[3] David Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, pages 258–268, Montreal, Quebec, Canada, 1998.

[4] David F. Bacon, Stephen J. Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, pages 111–132, Màlaga, Spain, June 2002.

[5] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA 98*, October 1998.

[6] Zoran Budimlić. *Compiling Java for High Performance and the Internet.* PhD thesis, Rice University, January 2001.

[7] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.

[8] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Design (OOPSLA)*, pages 1–19, 1999.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1993.

[10] Robert Costanza, Alexy Voinov, Roelof Boumans, Helena Voinov, Thomas Maxwell, Ferdinando Villa, and Josh Farley. PLM - Patuxent Landscape Model [WWW document]. `http://www.uvm.edu/giee/PLM`, 1998. (2002, October 10).

[11] U.S. Department of Defense. *High Level Architecture Interface Specification, Version 1.3*, February 1998.

[12] Julian Dolby and Andrew Chien. An automatic object inlining optimization and its evaluation. *ACM SIGPLAN Notices*, 35(5):345–357, 2000.

[13] Daniel Edelson. Smart pointers: They're smart but they're not pointers. In *Proceedings of the 1992 Usenix C++ Conference*, pages 1–19, August 1992.

[14] J. Epstein and R Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. MIT Press, 1996.

[15] Michael D. Fischer. Computer-based simulation modeling for anthropologists [WWW document]. `www.era.anthropology.ac.uk/Era_Resources/Era/Simulate/index.html`. (10 October 2002).

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, October 1994.

[17] G. N. Gilbert and J. Doran (ed). *Simulating Societies: The Computer Simulation of Social Processes*. UCL Press, London, 1993.

[18] P. P. Goncalves and P. M. Diogo. Geographic information systems and cellular automata: A new approach to forest fire simulation. In *Proceedings of Fifth European Conference and Exhibition on Geographic Information Systems, EGIS '94*, volume 1, pages 603–617, Utrecht, 1994. EGIS Foundation.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 1997.

[20] Dirk Grunwald and Benjamin G. Zorn. Customalloc: Efficient synthesized memory allocators. *Software - Practice and Experience*, 23(8):851–869, 1993.

[21] Java Grande Numerics Working Group. Improving Java for numerical computation [WWW document]. `math.nist.gov/javanumerics/reports/jgfnwg-01.html`. (10 October 2002).

[22] Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, N. Shenoy, and Prithviraj Banerjee. Enhancing spatial locality via data layout optimizations. In *European Conference on Parallel Processing*, pages 422–434, 1998.

[23] Thomas Kistler and Michael Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):490–505, May 2000.

[24] Keunwoo Lee. Exploded objects: A representation optimization for object-oriented languages. Ph.D Qualifying Examination Report, University of Washington Department of Computer Science and Engineering, 2001.

[25] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM))*, pages 358–367, 1998.

[26] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition.* Addison Wesley, 1999. Available at `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[27] James E. Marca, Craig R Rindt, and Michael G. McNally. A simulation framework and environment for activity based transportation modeling. Technical Report AS-WP-99-2, University of California, Irvine, November 1999.

[28] Elke Mentges. Concepts for an agent-based framework for interdisciplinary social science simulation. *Journal of Artificial Societies and Social Simulation*, 2(2), April 1999.

[29] Michael Möhring. Social science multilevel simulation with MIMOSE. In Klaus G. Troitzsch, Ulrich Mueller, G. Nigel Gilbert, and Jim E. Doran, editors, *Social Science Microsimulation.* Springer-Verlag, Berlin, 1995.

[30] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. `http://www.swarm.org/archive/overview.ps`, June 1996.

[31] Michael Noth. Centralia: A language and software architecture for multi-agent microsimulation (a preliminary specification). Available at `www.cs.washington.edu/homes/noth/urbansim/lang/Lang.pdf`.

[32] Michael Noth, Alan Borning, and Paul Waddell. An extensible, modular architecture for simulating urban development, transportation, and environmental impacts. *Computers, Environment and Urban Systems*, 27(2):181–203, March 2003.

[33] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, Warsaw, Poland, April 2003.

[34] Object Technology International, Inc. Eclipse platform technical overview, February 2003. Available at `www.eclipse.org/whitepapers/eclipse-overview.pdf`.

[35] R. J. Pryor, N. Basu, and T. Quint. Development of Aspen: A microanalytic simulation model of the U.S. economy. Technical Report SAND96-0434, Sandia National Labs, 1996.

[36] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, 1977.

[37] W. Shen, D. Xue, and D. H. Norrie. An agent-based manufacturing enterprise infrastructure for distributed integrated intelligent manufacturing systems. In Hyacinth S. Nwana and Divine T. Ndumu, editors, *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-98)*, pages 533–548, London, UK, 1998.

[38] Leigh Tesfatsion. Agent-based computational economics (ACE) [WWW document]. `www.econ.iastate.edu/faculty/tesfatsion/ace.htm`. (10 October 2002).

[39] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON*, 1999.

[40] José M. Vidal and Ed Durfee. Building agent models in economic societies of agents. In Milind Tambe and Piotr Gmytrasiewicz, editors, *Working Notes of the AAAI-96 Workshop on Agent Modeling*, pages 90–97, Portland, OR, 1996.

[41] Paul Waddell, Alan Borning, Michael Noth, Nathan Freier, Michael Becke, and Gudmundur Ulfarsson. Microsimulation of urban development and location choices: Design and implementation of UrbanSim. *Networks and Spatial Economics*, 3(1):43–67, 2003.

[42] Jon L. White. Advanced CLOS and meta-object protocols (abstract). In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, page 220. ACM Press, 1992.

[43] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proc. ACM Symposium on Programming Language Design and Implementation*, pages 30–44, June 1991.

[44] Jongwook Woo, Isabelle Attali, Denis Caromel, Jean-Luc Gaudiot, and Andrew L. Wendelborn. Alias analysis on type inference for class hierarchy in Java. In *Proceedings of the 24th Australasian conference on Computer Science*, pages 206–214, Gold Coast, Queensland, Australia, 2001.

Appendix A

POLYGLOT-SPECIFIC PROTOTYPE IMPLEMENTATION DETAILS

This appendix describes how the prototype implementation was developed as an extension to the Polyglot extensible Java compiler [33]. Polyglot uses abstract syntax trees as an internal representation of programs. Compiler passes consist of one or more visitors [16] applied to an AST that perform analyses and/or return partially- or wholly-modified ASTs as a result.

The extension supporting exploded objects consists of roughly $27,000$ lines of Java code and $7,500$ lines of templates for specialized data types.

An extension or modification to Java can be written as an extension to Polyglot, which typically encompasses modifications to the parser, creation of any new classes used to represent abstract syntax tree nodes for the extension, modification of the type system, changes to the set of compiler passes that operate over the abstract syntax tree, changes to code generation, and creation of any other supporting code. The prototype implementation of exploded objects involves work on, or consideration of, all of these areas. The following sections describe each of these areas, as well as some of the more significant challenges encountered during implementation.

A.1 Polyglot Parser Modifications

The only modification to the parsing process needed to support exploded objects is the addition of the `exploded` tag. All other aspects of the exploded object translation process, such as making use of specialized data types, etc., involve user code making use of normal Java language constructs that are processed by the translation engine. A separate (simple) parser is used for processing templates for specialized types.

A.2 New Abstract Syntax Tree Nodes

No new types of AST nodes were required for the prototype implementation. Instead, existing flag information associated with certain AST nodes has been augmented to include a flag indicating that the type is an exploded type.

A.3 Type System Modifications

Modifications to the type system to support exploded objects includes the addition of an **`exploded`** tag that can be associated with class-type objects and ensuring that the typing rules involving exploded objects are obeyed. The latter is encompassed by several new type-checking passes, as described in Section A.4.2, and by augmenting the normal Java typechecker as found in Polyglot to ensure that no assignments between exploded and non-exploded objects are performed, and that no exploded objects are passed as actual parameters where a normal Java object is required.[1]

A.4 Compiler Passes and Visitors

One of Polyglot's features that facilitates its use for experimenting with extensions to Java is its representation of compiler passes as the composition of one or more visitors applied to an abstract syntax tree. Visitors perform AST analyses, and can perform modifications to the AST[2] as part of the process.

The prototype implementation of the exploded object transformation process includes a number of new compiler passes. Passes have been added to perform analyses, augment the type-checking process, generate specialized data structures, perform translations of code using exploded objects into pure Java code suitable for separate compilation, and perform other incremental modifications to the AST to simplify other passes or analyses.

---

[1] e.g., when a parameter of type `Object` is required; this does not preclude the passing of exploded objects to methods that accept exploded objects, including passing subclasses of exploded objects where the formal parameter's type is a superclass of the exploded object type being passed as an actual parameter.

[2] Each visitor returns an entire AST, which is either the original unchanged AST or a modified copy of the original; strictly speaking, visitors do not perform in-place modifications of the original AST.

A.4.1 Analysis Visitors

Apart from type-checking visitors described below, the prototype implementation includes visitors that gather information on possibly-aliasable types that are involved with exploded object types, construct a first approximation of aliasing information between references, store inheritance relationships between exploded object types for later usage, and determine all templates that are involved in the data structure specialization process. Of these, only the last is used to any significant extent in the prototype; the others are intended to be used with possible future extensions of the prototype.

A.4.2 Type-Checking Visitors

The only special typechecking rules required in the prototype implementation are embodied in several visitors. One visitor examines the inheritance hierarchy of exploded objects to ensure that no exploded object type extends or is extended by a non-exploded type, in the process of determining which exploded object types require run-time type information to be associated with them. Another visitor ensures that no interfaces are implemented by exploded object types while determining the full set of class and interface types that may be involved in the exploded object translation process. It also verifies that any methods invoked on an exploded object refer to actual methods of the exploded object's type or its supertypes and not any that would normally be inherited from Java's `Object` type. This includes ensuring that any implicit calls to `toString` are converted to explicit calls, so that other aspects of the translation process are performed properly.

To see why this particular visitor is required, consider the effect of failing to apply the `toString`-conversion visitor to the code in the top part of Figure A.1. Without introducing an explicit `toString` invocation where the implicit invocation was present in the original Java version, just the **`int`** index of the `Household` object will be printed, instead of its textual representation. The correct translation, with an explicit call to `toString`, is shown in the bottom portion of the figure. A separate visitor is used to ensure that no exploded object instances are used for synchronization purposes. In particular, it ensures that no expression with an exploded object type is used in a **`synchronized`** statement and no non-static **`synchronized`** methods appear in an exploded object's definition.

```
1 // Original code
2 Household hh = ...;
3
4 System.out.println("The household is " + hh + ".");
5
6 // Naive translation approach that produces incorrect results
7 int hh_idx = ...;
8
9 System.out.println("The household is " + hh_idx + ".");
10
11 // Correct translation that requires the implicit toString call to be
12 // introduced
13 int hh_idx = ...;
14
15 System.out.println("The household is "
16   + CanonicalSet_Household.toString(hh_idx) + ".");
```

Figure A.1: Illustration of why it is essential to wrap all implicit calls to the `toString` method with explicit calls when exploded object types are involved. The naive translation shown in the middle part of the figure will result in a simple **`int`** representing the index of the `Household hh` being printed instead of the textual description. The correct translation is shown at the bottom of the figure.

A.4.3 Data Structure Specialization Visitors

Several visitors are used in the prototype implementation to generate specialized data structures through template instantiation. Different visitors are used to generate the intermediate and final forms of both specialized data structures and canonical sets. Of the different visitors, the most complex is the one that instantiates the final version of canonical sets that contain all exploded object instances.

In addition to token-based substitution, the final-form canonical set visitor builds customized versions of the `create` method to create a new instance of an exploded type by weaving together constructors from the exploded type and its supertypes, as well as any code called through implicit or explicit chained constructor calls (e.g., calling `this()` in one constructor to invoke another). The visitor also generates modified versions of all methods from the exploded type and its supertypes that take into account that `this` in the context of the method's body has become an index into parallel arrays contained within the canonical set, and that fields, variables, arguments, and return types of an exploded object type must be converted into indices into arrays, possibly including run-time type information.

Intermediate forms of specialized data structures are used for testing the application before the final conversion to exploded object form is performed, and are generally just wrappers around normal Java collections. The final forms of specialized data structures are much more involved, typically being complete implementations of an abstract data type that is specialized to handle one or more Java primitives, such as holding `int` and `byte` pairs to represent a reference to an exploded object that requires run-time type information. Chapter 7 provides more information about the structure and use of the templates.

A.4.4 Translation Visitors

The translation process that converts code involving exploded objects into normal Java code is divided into three main parts: generating intermediate versions of exploded object types for testing purposes, processing internal references to intermediate forms of exploded object types into final forms, and translating all references to exploded object types into code that makes use of canonical sets.

The simplest of the translation visitors is the one that generates intermediate versions of exploded object types, which are provided to assist users in adapting their code to make use of canonical sets and specialized data structures. For the intermediate version, an exploded object type is exactly the same as the original Java version. The translation process for the intermediate form simply strips off the **exploded** flag and passes the resulting ordinary Java class type to the normal Java code generation visitor as provided by Polyglot.

As described in Appendix B, the intermediate version of generated code is used in the translation process to generate the final version. In order for the intermediate version of the generated code to be properly typechecked, the Polyglot compiler needs to be able to load the intermediate forms of exploded objects and specialized data types, which are normal (non-exploded) Java object forms. After the initial stages of loading and typechecking, therefore, the abstract syntax trees for user-provided code contain references to normal Java types and not to exploded types which will have been loaded in conjunction with the intermediate version of the application. For translation to function properly, a visitor is used to replace all internal references to the Java object version of exploded objects with a reference to the internal representation of the final exploded form. This is required so that subsequent analyses and translation passes can perform Java-level equality tests involving only the exploded forms of objects, and not need to continually check to see if each internal reference to a Java object is actually an internal reference to the intermediate version of an exploded object.

For example, an application that has an exploded type of `Household` will have two distinct internal Polyglot type nodes representing a "`Household`", one for the normal Java object version generated for use in the intermediate testing stage, and one for the exploded object version. By replacing all internal references to the Java object version with the exploded object version, other passes can be much simpler in design.

The most complex component of the prototype implementation is the set of visitors that perform translation of code involving references to exploded objects into its final form that makes use of canonical sets. These visitors must identify every part of every branch of an AST that makes use of a reference to an exploded type in any way, and perform the appropriate transformations to translate the branch into normal Java code. The bulk of this complexity arises from three sources: the necessity for non-local changes, handling the large numbers of special cases for certain forms,

and programmatically generating the translated code.

For some aspects of translation, abstract syntax tree transformations are entirely local and reasonably straightforward in nature, involving replacing a single node (or branch) in the abstract syntax tree with another node (or branch). For example, a reference to a static field in an exploded object type is converted to a reference to a static field in the exploded object type's canonical set, as show in Figure A.2. Other forms of translation involve non-local changes, such as replacing a local variable to an exploded object with a pair of local variables used to store the exploded object's index and run-time type, or replacing a simple method call with a **`switch`** statement. Figure A.3 illustrates these forms of non-local translation. The added complexity in this form of translation arises from the replacement of one original AST node with multiple nodes in the AST, such as replacing one local variable declaration node with two, or replacing a single call node with nodes that declare a local temporary variable and a **`switch`** block to initialize it.

The second source of complexity in the translation-oriented visitors arises from the large number of cases that the translation process must handle. References to exploded object types can arise in many different contexts, such as on either side of an assignment, as the target of or a parameter to a call, as fields or local variables, as array types, return types, or as the implicit **`this`** or **`super`** in the body of a method. Each of these cases must be handled appropriately, and many of these top-level cases have numerous sub-cases which involve moderately or substantially different manipulations of internal compiler data structures to identify and translate properly.

For example, consider a field usage of the form `foo.bar`. The field target `foo` can legally be any one of another field usage, a local variable, a formal parameter, an array access, **`this`** or **`super`**, or a method call. If the field target is an exploded type, it must be translated differently than a normal Java object as the field target, namely by converting it into an access into the appropriate canonical set. If the field target is an exploded object type that requires run-time type information, additional code must be added to ensure that the appropriate version of the field `bar` is accessed based on the static type of the target, in accordance with §8.3.3.2 of the Java Language Specification [19].

For each of these cases, the nature of the field `bar` itself increases the complexity. If the field is an exploded type, the field access needs to be translated into an access into the appropriate canonical set. If the field type requires run-time type information, the field access must add at least one additional statement that handles the run-time type information. Additional translation steps may

```
1 // Original version
2 public exploded class Household {
3 ...
4   public static final byte INCOMECATEGORY_LOW = 1;
5 ...
6 }
7
8 // Original form of static field reference, in some other class
9 byte incCat = Household.INCOMECATEGORY_LOW;
10
11 // Translation of the original version
12 public class CanonicalSet_Household {
13 ...
14   public static final byte INCOMECATEGORY_LOW = 1;
15 ...
16 }
17
18 byte incCat = CanonicalSet_Household.INCOMECATEGORY_LOW;
```

Figure A.2: Translation of a reference to a static field in an exploded object type. The top portion shows the original code within the `Household` type and the original usage, and the bottom portion shows the translated version in the canonical set for the `Household` exploded type and the resulting version of the reference.

```
1 // Original code.
2 Person somePerson = ...;
3 ...
4 Person p = somePerson;
5 String personDesc = p.toString();
6
7 // Translation of original code.
8 int somePerson_idx = ...;
9 byte somePerson_rtt = ...;
10 ...
11 int p_idx = somePerson_idx;
12 byte p_rtt = somePerson_rtt;
13
14 String tmp$0;
15 switch ( p_rtt ) {
16   case RunTimeTypes.PERSON_RTT:
17     tmp$0 = CanonicalSet_Person.ref.toString(p_idx);
18     break;
19   case RunTimeTypes.EMPLOYEE_RTT:
20     tmp$0 = CanonicalSet_Employee.ref.toString(p_rtt);
21     break;
22   default:
23     throw new RuntimeException(...);
24 }
25 String personDesc = tmp$0;
```

Figure A.3: Translation of a local variable declaration, and subsequent method invocation using that variable, for an exploded object type that requires run-time type information. For this example, assume that there are `Employee` and `Person` exploded object types, where `Employee` is a subclass of `Person` and both have separate versions of the `toString` method.

be required if the field `bar` is actually an array of exploded object instances, with more complexity introduced if the exploded object type involved requires run-time type information.

In addition, the context in which the field usage appears can affect translation. For example, if the field usage appears as an actual parameter to a method call, the method call may need to have an extra parameter added to handle the run-time type if the field is of an exploded object type that requires run-time type information. The field usage's position in an assignment (left-hand side versus right-hand side) can also affect the translation process.

While in practice, some of the numerous combinations of special cases as described above can be collapsed, and the AST-flattening visitor pass described in Section A.4.5 simplifies some of the cases, many are just different enough to require separate code to handle each combination. This complexity often arises from the need to handle different internal Polyglot data structures in subtly different fashions based on the underlying AST node types involved.

It is important to note that much of the above complexity could have been avoided by choosing a different internal representation for use in the translation process, such as the three-address form for Java bytecodes as used in Soot [39]. However, additional efforts would have been required to convert from the AST-based representation used by Polyglot to the alternate form, and back again, as well as possibly requiring additional efforts to handle code generation involving a simplified internal representation. Using an alternate internal representation would have negated the advantages to using the Polyglot environment, which was deemed an unacceptable consequence for the purposes of the prototype implementation.

The third source of complexity in the translation-oriented visitors comes from programmatically generating non-trivial replacement nodes for insertion into the abstract syntax tree. For many aspects of translation, a simple AST node such as an assignment or call must be replaced by multiple non-trivial statements, such as the translation of the `toString` invocation shown in Figure A.3. As the results of the translation process may be fed to other AST visitors, such as the code generation visitor, appropriate AST subtree replacements must be constructed programmatically. While the process of generating new AST nodes and connecting them together using Polyglot is relatively straightforward, there are often subtleties as to ordering of node creation, assignment of types, etc., that can combine to complicate the task.

A.4.5 Other Visitors

The prototype implementation contains several other visitors that perform various tasks designed to simplify other analysis and translation passes. These visitors include one that wraps all `if-then` statements with code blocks, one that separates field and variable initializations into separate declaration and initialization statements, and a visitor that flattens complex expressions in the AST by introducing temporary variables.

Internal Polyglot representations of `if-then` statements may contain a single statement or a block of statements associated with the `then` or `else` conditions. To simplify the translation process, a visitor converts all single-statement results with a block containing the single statement. This allows translation-related visitors that must replace one statement with multiple statements to always assume that there is a block of code into which their translated statements can be placed. Figure A.4 illustrates the effects of this visitor.

Another visitor replaces all field and local variable declarations that have initializers with separate uninitialized declarations and an assignment to initialize them. `static` fields have their initializers moved into `static` blocks. `final` fields are unaffected.[3] Initializers of instance fields are moved into each constructor call, so that they are initialized before the constructor's body executes as described in §15.9.4 of the Java Language Specification [19]. The simplification performed by this visitor allows other analyses and the translation process to not need to deal with the extra complexity of handling initializers. Figures A.5 and A.6 illustrate the effects of this visitor. Note that when an instance field is initialized outside of a constructor and then assigned to inside a constructor, code generated by this visitor will result in a dead assignment which would presumably be eliminated by an optimizing compiler. The initialization of the `salary` field in Figure A.6 is an example of the generation of this sort of dead assignment.

A third visitor is used to partially flatten the abstract syntax tree, replacing complex expressions with simpler ones by introducing local temporary variables. Flattening the AST greatly reduces the number of special cases that must be handled by the analysis and translation passes, as well as

---

[3]Note that due to the ordering of class loading and initialization, it is not generally possible to ensure that full initialization has taken place before invoking a `create` method to create an instance of an exploded object type to initialize a `static final` field. However, this is not a significant limitation in practice as it is considered bad form to use non-primitive types as `static final` fields.

```
1 // Original code
2 if ( someExpression )
3   System.out.println("Expression is true");
4 else
5   System.out.println("Expression is false");
6
7 // Code after visitor wraps singleton statements
8 if ( someExpression ) {
9   System.out.println("Expression is true");
10 } else {
11   System.out.println("Expression is false");
12 }
```

Figure A.4: Illustration of the effects of the **if-then** statement-wrapping visitor. **if-then** statements with singleton statements for the **then** or **else** portions have the singleton statement replaced by a block of code containing the statement.

simplifies the cases that remain. For example, calls that have non-trivial targets are broken apart into a temporary local variable for the target and a simpler call expression involving that temporary. Non-trivial expressions passed as arguments to procedures are replaced with local variables initialized with the expression. Complex field accesses are broken apart into simpler accesses involving a local temporary and a field access. Figures A.8 and A.9 show an example of the effects of the AST-flattening visitor on the code in Figure A.7. The introduction of temporary variables does introduce a small performance penalty, but a simple experiment in which all temporary variables present in the final translated version of an application were manually replaced by their original expressions resulted in a performance gain of only 2%-4%. The impact of eliminating temporary variables may be considerably greater for certain types of code, for example if the code's structure causes large numbers of temporary variables to be created inside inner loops.

```
1 public class Employee {
2   public static final byte POSITION_NONE = 1;
3   public static final byte POSITION_ENTRYLEVEL = 1;
4   public static final byte POSITION_JUNIOR = 2;
5   ...
6
7   public static float MINIMUM_WAGE = Economy.getMinimumWage();
8
9   protected Job lastJob = null;
10  protected float salary = 0.0f;
11  protected byte positionCode;
12
13  public Employee() { positionCode = POSITION_NONE; }
14
15  public Employee(byte position, float salary) {
16    this.salary = salary;
17    positionCode = position;
18  }
19 }
```

Figure A.5: Original version of code prior to application of the visitor that breaks up declarations with initializations into separate uninitialized declarations followed by initializing assignments. Assume that the `Job` and `Economy` classes are defined elsewhere. Figure A.6 shows the effect of the visitor on this code.

```
1 public class Employee {
2   public static final byte POSITION_NONE = 1;
3   public static final byte POSITION_ENTRYLEVEL = 1;
4   public static final byte POSITION_JUNIOR = 2;
5   ...
6
7   public static float MINIMUM_WAGE;
8   static {
9     MINIMUM_WAGE = Economy.getMinimumWage();
10  }
11
12  protected Job lastJob;
13  protected float salary;
14  protected byte positionCode;
15
16  public Employee() {
17    this.lastJob = null;
18    this.salary = 0.0f;
19    positionCode = POSITION_NONE;
20  }
21
22  public Employee(byte position, float salary) {
23    this.lastJob = null;
24    this.salary = 0.0f;
25    this.salary = salary;
26    positionCode = position;
27  }
28 }
```

Figure A.6: Translated code from Figure A.5 showing the effects of the visitor that separates declarations with initializers into uninitialized declarations followed by initializing assignments. Note that **`static`** variables are initialized in static code blocks added at the top level of the class, non-static initializations are moved into separate blocks within constructor definitions, and **`final`** variables are not affected.

```
1 public class LinkedListNode {
2   public String data;
3   public LinkedListNode next;
4
5   public LinkedListNode(String d) { next = null; data = d; }
6
7   public LinkedListNode getNextNext() {
8     if ( next != null ) {
9       return next.next;
10    }
11  }
12  public String toString() {
13    return new String("(" + data + ", "
14      + ((next == null) ? "[null]" : next.toString());
15  }
16 }
17
18 LinkedListNode lln1 = new LinkedListNode("One");
19 lln1.next = new LinkedListNode("Two");
20 lln1.next.next = new LinkedListNode("Three");
21 lln1.next.next.next = new LinkedListNode("Four");
22
23 LinkedListNode lln2 = lln1.next.getNextNext();
24
25 System.out.println("lln1 chain is " + lln1.toString());
```

Figure A.7: Original version of some example code prior to application of the AST-flattening visitor. Figures A.8 and A.9 illustrate the effect of the visitor on the above code.

```
1 public class LinkedListNode {
2   public String data;
3   public LinkedListNode next;
4
5   public LinkedListNode(String d) { next = null; data = d; }
6
7   public LinkedListNode getNextNext() {
8     if ( next != null ) {
9       LinkedListNode tmp$0;
10      tmp$0 = this.next;
11      return tmp$0.next;
12    }
13  }
14  public String toString() {
15    String tmp$1;
16    tmp$1 = next.toString();
17    String tmp$2;
18    tmp$2 = ((next == null) ? "[null]" : tmp$1;
19    String tmp$3;
20    tmp$3 = "(" + data + ", " + tmp$2 + ")";
21    return new String(tmp$3);
22  }
23 }
```

Figure A.8: Part one of an illustration of the AST-flattening visitor's effects on the code from Figure A.7. Note how complex field accesses and call targets are replaced with local variables, to reduce the number of special cases other analyses and translation passes must handle.

```
24 LinkedListNode lln1 = new LinkedListNode("One");
25 lln1.next = new LinkedListNode("Two");
26 LinkedListNode tmp$4;
27 tmp$4 = lln1.next;
28 tmp$4.next = new LinkedListNode("Three");
29 LinkedListNode tmp$5;
30 tmp$5 = tmp$4.next;
31 tmp$5.next = new LinkedListNode("Four");
32
33 LinkedListNode tmp$6;
34 tmp$6 = lln1.next;
35 LinkedListNode lln2 = tmp$6.getNextNext();
36
37 String tmp$7 = lln1.toString();
38 String tmp$8 = "lln1 chain is " + tmp$7;
39 System.out.println(tmp$8);
```

Figure A.9: Part two of an illustration of the AST-flattening visitor's effects on the code from Figure A.7. Note how complex field accesses and call targets are replaced with local variables, to reduce the number of special cases other analyses and translation passes must handle.

A.5 Changes to Code Generation

Considerable effort has been undertaken to not require any changes to the code generation process. Polyglot provides a visitor that generates Java code based on a traversal of an abstract syntax tree, so the translation process for exploded objects involves processing and transforming ASTs into a final form which involves only normal Java constructs. This allows the default Polyglot code generation visitor to be used for the bulk of code generation.

Separate visitors are used to instantiate templates for specialized data types. These visitors rely on a combination of token-level substitution in copied text and on modifying AST branches to convert them into a form suitable for being passed to the default Polyglot code generation visitor. An example of token-level substitution is the simple replacement of the `<TYPENAME>` tag with the name of the type for which instantiation is being performed. An example of the latter is the generation of an AST for each `create` method in the canonical set for each exploded object type, based on assembling and manipulating the `AST` sub-trees associated with each constructor in the original exploded object type and its supertypes.

A.6 Supporting Code

The prototype implementation of the approach to exploded objects as described in this text includes a variety of supporting code. Logging facilities are provided, as well as handling of extended command-line options for the exploded object extension to Polyglot, utilities to assist with name-mangling, creation of temporary variables, conversions between different internal Polyglot types, and code to generate the `RunTimeTypes` file that contains run-time type constants for use with exploded objects that require run-time type information.

A simple logging class is provided. It allows for writing to standard output or a file, and supports automatic indentation and line-wrapping.

As is required for all Polyglot extensions, a specialized version of the `ExtensionInfo` Polyglot class is part of the prototype implementation. It contains code for managing the various extension-specific command line parameters used to indicate the target package to which generated code should be written, where the debugging flag file is found, etc.

To help with common tasks, some utility methods are included in a helper class. They provide

automatic name-mangling code (e.g., to convert an exploded type instance name `hh` to `hh_idx` and `hh_rtt` for the exploded object index and run-time type information portions), methods for generating a declaration of and reference to the next uniquely-named temporary variable of the provided type, conversions between several commonly-used Polyglot internal types, etc.

The `RunTimeTypes` class contains a list of named constants for each of the exploded object types present in an application. It also contains helper methods to convert from a run-time type to a textual description for error-reporting purposes.

Appendix B

USING THE PROTOTYPE IMPLEMENTATION

This appendix provides usage information for the prototype implementation. The prototype implementation is designed to be used in a multi-step fashion, allowing for testing of code modifications prior to the full translation process. It provides support for placing generated code (intermediate and final versions) into a package of one's choosing, easy configuration of debugging and logging options to examine the inner workings of the translation process, and a simple harness for executing the different steps of generation, compilation, translation, and execution.

B.1 Converting an Application to Use Exploded Objects

The general process for converting an application to use exploded objects is as follows. Each step is described in more detail in subsequent sections.

1. Identify candidate object types for conversion to exploded objects.

2. Tag candidate object types with the **`exploded`** keyword.

3. Generate the first version of specialized data types

4. Modify application code to make use of specialized data types

5. Test the modified application code to ensure that modification was successful

6. Generate full translation of modified application code

7. Compile and execute fully-translated version of application

B.1.1 Identify Candidate Objects for Conversion

The most important step in applying exploded objects to an application is to identify good candidate objects for the conversion process. As described previously, a prime candidate for conversion into an exploded object is a type that has the following properties:

- Is present in large numbers in the simulation

  Tends to be created in large quantities at infrequent intervals

- Tends to be long-lived

- Has a significant number of fields that are be Java primitive types such as **`byte`** or **`int`**

- Is removed from the simulation environment in a well-defined manner (not just garbage-collected)

- Is not used for synchronization or reflection

Only the last two properties (well-defined removal and no synchronization or reflection) are absolute requirements for conversion to exploded object form. In general, the more the other properties apply to the type, the more benefits may be gained from conversion to exploded object form.

B.1.2 Tag Candidate Object Types

Once one or more candidate object types have been identified, the translation process can begin. All `.java` code dealing with exploded objects is renamed to have an `.eo` extension.[1] Candidate object types are tagged as being exploded by adding the **`exploded`** keyword to their definition, as shown in Figure 5.1.

B.1.3 Generate Specialized Data Types

After candidate object types have been tagged with the **`exploded`** keyword, specialized data types and canonical sets can be generated. The prototype implementation's compiler can be run in gen-

[1]This step is not strictly necessary, but is useful for differentiating exploded object-related code from other code. It is perfectly acceptable to rename all `.java` files of an application to `.eo` regardless of their usage of exploded objects.

eration mode, which generates intermediate versions of specialized types. Intermediate specialized versions still operate on normal Java objects, but have the same semantics as the fully-translated specialized versions. For example, the intermediate version of a canonical set is a wrapper for an array of "live" Java object instances of what will become exploded object instances, and the intermediate version of a `HashSet` is a wrapper of the `java.util.HashSet` collection. These intermediate specialized versions are used for testing purposes and to help with the conversion to usage of exploded objects.

B.1.4 Modify Application to Use Specialized Data Types

At this point, the application is manually modified to make use of specialized data types including canonical sets, and any other modifications as required to reflect the altered semantics of exploded objects. For example, all object allocations via **new** for exploded object types are replaced with an invocation of the `create` method from the appropriate canonical set, and a `java.util.HashSet` is replaced by a specialized `HashSet`, such as `HashSet_Household` for holding `Household` instances. Explicit deallocation of exploded object instances is added during this step as well, by inserting calls to the `destroy` method of the appropriate canonical set in the same locations where instances of the object are normally removed from the simulation. For example, if `Household` objects are removed from a simulation when they move out of a city, the `destroy` method of the `Household`'s canonical set should be invoked after any bookkeeping involving the `Household` is performed. Package imports may need to be adjusted to ensure that the specialized data types can be located.

B.1.5 Test Modified Application

Once the application has been modified to make use of the specialized data types and canonical sets, it should be compiled and tested to ensure that it still works as expected. As this step still involves normal Java objects at all stages, performance and memory usage will be the same as or somewhat worse than the original unmodified application. This should not be a concern, as this step is for testing of correctness, not performance.

B.1.6 Generate Full Translation

Once testing of the intermediate version has been completed, the full translation process is executed on the modified version of the application. Pure Java code is produced as output.

B.1.7 Run Fully-Translated Application

The last step in the conversion process is to compile the fully-translated form of the application and execute it. As the translated version of the application is pure Java code, any Java compiler and run-time environment can be used for this step.

B.2 Working with Packages

The prototype implementation allows for specification of target packages into which intermediate and final versions of specialized types and canonical sets are to be written. The target packages for generated intermediate and translated final versions of specialized types are specified via command-line parameters passed to the compiler. If a package is supplied, it is appended to the package designation, if any, of the original exploded type for which specialized collections are being generated. For example, if an exploded type `Household` were originally found in the `test.urbansimlet` package, and a target package of `gencode` were specified for generated code, specialized collections of `Households` would be placed in the `test.urbansimlet.gencode` package.

Any imports specified within an exploded type's original definition are copied into the definitions of all specialized code for that type.

One ramification of this approach is that it may be necessary to explicitly specify imports of classes from the same or higher levels of the package hierarchy in an exploded type that makes use of them, even if the original Java version did not require an explicit import. For example, if a `Household` exploded type makes use of a `House` type that are both at the same level of the package hierarchy (e.g., `test.urbansimlet`), then the `Household` exploded type should explicitly import `test.urbansimlet.House`. This is required as the specialized collections of `Households` will be located in package `test.urbansimlet.gencode` for example, which does not automatically have access to all types declared in the `test.urbansimlet` package. In practice, it is trivial to add the necessary import(s) while making other modifications to use specialized data types.

B.3 Debugging and Logging Options

A simple text file contains a number of debugging and logging flags that allow for tuning output of the generation and translation process. The options include specifying the amount of detail written to a log file concerning each separate stage of the generation and translation processes, controlling whether intermediate versions of the abstract syntax tree for each type being processed are written, etc. Output produced by enabling these options can be helpful when modifying or adding new specialized type templates, modifying the prototype implementation, tracing through the translation path that leads to specific forms of translated code being generated, or tracking down bugs in the prototype if any are suspected.

The debugging flag file is specified as a command-line parameter to the compiler, and if omitted, it is assumed that all flags are disabled so no extra debugging or logging information will be produced.

B.4 Translation Harness

A simple harness in the form of a shell script is supplied to help with the multi-step translation process. It assumes that all application code relating to or using exploded object types resides in a single directory, with other code located on the classpath. The harness can be run in one of seven modes, as described in Table B.1. Execution of the harness requires three parameters. The first parameter is the base directory of the project to be translated. The second parameter specifies the mode. The third parameter specifies the class name in which a `main` method can be found for execution purposes.

For applications with a more complex package structure, the translation harness can serve as a basis from which to construct a suitable script to drive translation.

B.5 Modifications to Converted Applications

The prototype implementation of the exploded object translation process is intended for use on stable applications as an optimization step, and not as part of a normal development process. However, it is not uncommon for additional modifications to be made to an already-converted application,

Table B.1: Execution modes for the translation harness supplied with the prototype implementation. It is assumed that the project is in directory (and package) *pd*, and the `main` method can be found in class *pMain*.

Mode	Description
`orig`	Compiles the code in *pd*`/orig`, executes the code by invoking the Java runtime environment on *pd*`.`*pMain*, captures the output, and displays the output or any compilation or execution errors.
`gen`	Generates the intermediate form of specialized data types based on `.eo` files located in *pd*`/gen`.
`gencomp`	Compiles the intermediate form of specialized data types from the `gen` stage.
`gent`	Generates the intermediate testing form of the application using the `.eo` files in *pd*`/gent`, where the `.eo` files have been modified to make use of specialized data types.
`gentrun`	Compiles the intermediate testing form of the application in the *pd*`/gent` directory, executes it by invoking the Java runtime environment on *pd*`.`*pMain* found in the *pd*`/gent` directory, captures the output, and then displays the output or any compilation or execution errors.
`trans`	Generates the final translated form of the application based on the `.eo` files in *pd*`/gent`, with the generated code being placed in a directory rooted at *pd*`/trans`.
`transrun`	Compiles the final translated form of the application as located in the *pd*`/trans` directory, executes it by invoking the Java runtime environment on *pd*`.`*pMain* found in the *pd*`/trans` directory, captures the output, and displays the output or any compilation or execution errors.

such as tagging additional object types for conversion to exploded form, replacing one specialized collection with another, or modifying compile-time constants.

As described previously, the translation harness script assumes that modified code from each step of the conversion process is stored in a separate directory. This makes it much easier to "back up" in the conversion process and re-run only the necessary phases without having to manually delete modifications to the original code base. For example, tagging an additional object type as `exploded` requires only incremental changes to the previously-modified code as needed to make use of new specialized types.

It is expected that a more robust version of the exploded object translation process would be required before it would be feasible to incorporate such into an active development process.

VITA

Michael E. Noth was born in Iowa City, Iowa on December 28, 1972. In 1991, he graduated as valedictorian from Iowa City City High School. He graduated *summa cum laude* from the University of Iowa in 1995 with a Bachelor of Science in Computer Science and a Bachelor of Arts in French. After working for CCAD at the University of Iowa for a year and a half designing and implementing driving simulator scenario editing and visualization software, he left Iowa for sunny Seattle. He received a Master of Science in Computer Science from the University of Washington in 1998, and a Ph. D. in Computer Science from the University of Washington in 2003.