SAFE SHARING OF OBJECTS IN A HIGH-PERFORMANCE PARALLEL LANGUAGE

Benedict Gomes*, Welf Löwe†, Jürgen W. Quittek‡, Boris Weissman*

{gomes, welf, quittek, borisv}@icsi.berkeley.edu

*. University of California at Berkeley and International Computer Science Institute
†. University of Karlshruhe, Germany and International Computer Science Institute
‡. NEC Labs, Berlin, Germany and International Computer Science Institute

International Computer Science Institute,
1947 Center St., Berkeley, CA 94704

Abstract. Consistency of objects in a concurrent computing environment is usually ensured by serializing all incoming method calls. However, for high performance parallel computing intra-object parallelism, i.e. concurrent execution of methods on an object, is desirable. Currently, languages supporting intra-object parallelism are based on object models that leave it to the programmer to ensure consistency.

We present the object model for Sather-2, that ensures object consistency while supporting intra-object concurrency thereby offering both safety and efficiency. The description starts with a simple and safe, but inefficient model and gradually increases the sophistication by introducing features for expressiveness and greater efficiency while maintaining safety.

A prototype implementation of Sather-2 is presented, along with an evaluation of the performance on a particle simulation. A discussion of the implementation and run-time measurements confirm the achieved efficiency, while code examples demonstrate the ease of expression.

1 INTRODUCTION

With the advent of commodity multi-processors, parallel hardware platforms are now more widely available than at any time in the past. Small and medium size shared memory multi-processors (SMPs) are now manufactured by most major hardware vendors. Furthermore, there is a plethora of applications that could benefit from this readily available multi-processor performance, ranging from mathematically-intensive simulations (the traditional users of high-performance systems), to databases, to web-servers. Programming these systems effectively requires a safe, yet efficient programming paradigm, in which potentially conflicting irregular activities are protected from each other without the introduction of excessive overhead. One approach, taken by languages such as Java [6][18], pSather [32], and C used with different thread libraries, is to provide the programmer with the primitives for parallelism (thread creation) and safety (locking, monitors etc.). This approach cannot provide any **guarantee** of safety to the programmer; in parallel contexts, large concurrent systems may contain race conditions that require horrific debugging efforts. Conflicting object accesses may be prevented by conservatively locking all object accesses, as is done in many of the Actor-style languages and some Active-Object languages. However, as we show in section 9, this results in excessive overhead[1].

1. Aggressive optimization may reduce this overhead, but such optimizations are typically complex to perform and fragile, leading to unpredictable performance.

One general problem with such object protection is that it may be excessively fine-grained; as noted by many others [13][3], programs usually naturally decompose into domains of related objects such that the objects within one domain do not interact significantly with the objects in other domains. However, modern object-oriented languages do not support this level of modularity - control over access to attribute variables may be expressed, but not to the state transitively reachable from the object (i. e. deep object state) [14][3].

We present the object model of Sather-2, a concurrent high-performance object-oriented language based on the sequential language Sather 1.1 [32], which supports object-modularity by controlling inter-domain aliasing. While other proposals for controlling object-modularity in sequential languages either prohibit inter-domain objects access [13], or force immutable semantics [20] (which may have unacceptable performance implications), we present a scheme for concurrent environments which allows the temporary sharing of internal state when such sharing cannot result in conflicting accesses. Finally, we also permit the explicitly marked transfer of objects between domains; the safety of such transfers may be verified by an optional run-time safety check.

We also describe a prototype implementation that demonstrates the performance of Sather-2. We use a simple version of particle simulation as our running example to illustrate language features, and we demonstrate that scalable high-performance is achieved. The same ideas should be directly applicable to other explicitly parallel object-oriented languages such as Java and Ada.

We develop our model starting from a conservative object model, by gradually adding the necessary features, maintaining object safety along the way.

- The remainder of this introduction first describes related work, both in the area of high-performance shared memory programming languages, and in the area of safety and object encapsulation and then describe our base sequential language, **Sather**.
- Section 2 describes a **particle simulation**, which we use as a running example to illustrate language points and also to demonstrate performance.
- Section 3 describes our notion of object safety and its basis in sequential consistency.
- Section 4 describes an object model which provide visitor/mutator based locking protection. We refer to a model in which all objects are protected in this manner as the **conservative** model.
- Section 5 introduces **shield** objects which provide protection in the face of concurrent access, and **interior** objects, which describe the interior state of shield objects and are protected by them.
- While interior objects will generally be visible only within their domain, we also permit them to be viewed from other domains when this cannot result in a loss of object safety. Section 6 describes how interior objects may be **temporarily transferred** between the object domains of shield objects in a manner that may be statically guaranteed to be safe.
- While temporary transfer permits many inter-domain interactions, in some cases objects must be permanently transferred between domains. Section 7 shows how interior objects may be **permanently transferred** between object domains, provided they are freed from their original domain.
- The annotations we introduce are an important part of the class interface. In order for safety to be preserved in the face of polymorphism, they must be respected during **subtyping**, as described in Section 8.
- Section 9 presents **performance results** on our example application.

1.1 Related Work

There are several strands of related work, that approach the problem of safe, high-performance concurrent programming from different angles. In general, the work has concentrated on either safety or high-performance, but rarely on both. We also review work done in the context of sequential languages with the aim of achieving true object encapsulation by controlling aliasing.

Efficiency

Some explicitly parallel object-oriented languages such as Sather 1.1 [32][11], Java [6][18], Ada [5], and C++ extended with thread and synchronization libraries such as POSIX threads [25] expose low-level details of memory and network consistency models to the programmer trading the simplicity of the underlying model for efficiency. The programmer is presented with a system-centric model since the low-level system optimizations are fully exposed to the programmer. In particular, such a model enables many software and hardware optimizations that can reorder instructions (or even program statements) to hide the latency of memory and network operations. While this model strives to achieve the highest possible efficiency, it may unduly compromise programmability by presenting the programmer with a fairly complex view of object states and transitions. Reasoning about parallel programs becomes cumbersome as the programmer needs to take into account the low-level reorderings that are enabled by the weak consistency model. Failure to do so results in hard-to-find data races. In addition, the portability of programs is by no means automatic - unless the development platform supports the weakest possible memory model, a program that compiles and runs correctly during development may display subtle data races when compiled for a platform with a weaker hardware memory model [10].

Safety at the Object Level

Many **Actor languages** [4] are based on a very simple programming model in which computation is performed by independent entities communicating by atomic non-blocking messages. The simplicity of the model however is achieved at the cost of expressiveness and efficiency: the requirement of the state update atomicity either introduces extra copying of local state or disallows intra-object parallelism; atomicity of all method invocation has a negative performance impact. The model usually relies on sophisticated compiler optimizations to limit the degree of parallelism in order to reduce queue management overhead on modern hardware [16]. On the positive side, the absence of intra-object parallelism and the atomicity of methods eliminates the programming overhead to ensure memory consistency. This, in turn, eliminates the source of pernicious bugs especially well known to programmers developing software for symmetric multiprocessors. Resulting programs are usually portable (although not necessarily efficient) across many parallel platforms. For some contexts, such as functional languages (base of some actors) certain performance and expressiveness limitations are well justified by the model simplicity and the absence of data races.

Expressing Object Grouping

In order to resolve the conflict between performance and safety, many programming models have resorted to controlling safety based on aggregates of objects.

Argus [20] introduced special guardian objects to control access to a set of resources composing the internal state of a guardian. Within a guardian full sharing of objects is allowed while no direct sharing of objects between guardians is permitted. In fact, the internal state of the guardians was built from standard (sequential) CLU objects [19]. The task of concurrency and safety management is entirely performed by the guardians. Guardians zealously protect their internal state from other guardians. If necessary, a guardian may create a copy of its internal objects and pass it to other guardians. Thus passing objects between guardians has a value semantics. Internal concurrency within a guardian is allowed and guardians are fully responsible for synchronizing their internal state. The model does not provide a static safety guarantee.

Similarly, Emerald [7] distinguishes between global and local objects at the implementation level. Emerald is a distributed system and global objects are allowed to move within the network. They also support remote method invocation. Local objects always remain within an enclosing object (i.e. the reference is never exported outside), cannot move on their own and do not support remote invocation.

Maintaining Group Encapsulation

In current object-oriented languages it is not possible to guarantee the encapsulation of a group of objects within a containing object. The reason is that aliases to the internal state may be erroneously released outside the containing object. While protection is provided for attribute *variables*, protection for the *state* that is transitively reachable from the object cannot be expressed at the language level. This problem has been recognized as one of the most serious challenges of object-oriented programming [14]. In spite of its importance, there have been only few proposals to ameliorate this deficiency. We will briefly look at a couple of proposals for sequential object-oriented languages.

Islands [13] provide a syntactic mechanism for the isolation of groups of objects based on richer argument and variable annotations. Bridges are protector objects which isolate interior objects (islands), by controlling the import or export of aliases to the interior objects. Absolute modularity is maintained through bridges. This is excessively restrictive for our purposes. Certain objects represent shared resources and it is natural that they be aliased between different domains (or islands). Furthermore, transfers between domains are not possible in this model. While the work on islands seeks to prevent all sharing of internal state between modules, some sort of object sharing is unavoidable in practical systems. The Island proposal is important methodologically in that it is specifically targeted at static safety *guarantees* rather than *optional annotations*.

In a similar vein, Balloon types [3] demonstrate how a similar degree of safety may be provided with fewer annotations and more sophisticated static analysis. However, balloon types only guarantee safety against static aliasing (i.e. aliases though state variables) and not against dynamic aliasing (aliasing through local variable that exist only while a function call is in progress)[2]. Since any aliasing between object groups, either static or dynamic, can potentially result in conflicting concurrent object accesses, balloon types are inadequate for providing safety in the face of concurrency. Furthermore, data may only be shared between balloon types through the use of copying which essentially limits the usefulness of this technique in high-performance domains.

The object model implemented by Sather 2.0 is similar in spirit to these approaches, but a different solution is necessitated by the different language goal of supporting safe and efficient *concurrent* programming. Unlike other design proposals, the object-model presented in this article has been actually implemented and a prototype compiler for Sather-2 has been built and used to obtain performance measurements on a number of applications and benchmarks as presented in later sections.

Hardware Issues: Memory Consistency Models

Memory consistency models supported by the modern multi-processors are often characterized by subtle, but important differences. Such weak consistency models include TSO and PSO [27], RSO [28], processor consistency [2], numerous flavors of release consistency [9], and other models supported only by the individual hardware vendors (DEC ALPHA [26], PowerPC [21], PentiumPro [15], etc.). As do many others [1][2][9][10], we believe that programmers should be presented with a single and simple programming model to shield them from the intricate details of the underlying hardware. Sequential consistency [17] is the most natural candidate for such a top level programming model. Sequential consistency is central to the Sather-2 notion of object safety. The great challenge is reconciling the high level sequentially consistent model with the weak consistency model supported by the hardware. The latter are responsible for up to 80% or performance improvements of modern microprocessors [10]. Unlike other approaches that concentrate on detection of deviations from sequentially consistent executions by developing program analysis tools [1], we provide hard guarantees of sequentially consistent execution on weak consistency hardware at the programming language level. We will further discuss the memory consistency issues and the interaction between the safety requirements and sequential consistency in section section 3.

1.2 Sather

The basis for Sather-2 is the sequential language Sather 1.1 [11][32]. We briefly introduce some of the features of Sather that are relevant to the rest of this article. Sather is a relatively standard, object-oriented language in the general style of Java, Eiffel or C++, but is specifically targeted at safe, high-performance computing.

Like most other object-oriented languages, Sather supports encapsulation by providing **privacy annotations**. Attributes are accessed by a pair of implicit accessor functions, with the appropriate privacy annotations. Thus the declaration

```
readonly attr a:INT; -- declaration
```

is associated with an implicit private writer method, used for assignment to ‘a’ and an implicit public reader method used for accessing the value of ‘a’:

```
private a(val:INT);  -- implicit writer method
a:INT;               -- implicit reader method
```

Sather separates the notion of **subtyping and code-inclusion**, in a manner similar to Java. Classes may be abstract (and contain no code) or concrete. Sub-typing and code inclusion are completely distinct mechanisms, and both

2. Protection against dynamic aliasing using "opaque" balloon types is mentioned in [3], but the details presented are only concerned with immutable objects.

multiple sub-typing as well as multiple code-inclusion are supported. Subtyping in Sather follows the standard statically type-safe rule of contravariant conformance (contravariant argument types and covariant return types).

We also provide **privacy on a per-object basis**; thus, methods in a class only have access to the private state of that object and not to the private state of other objects of the same class.

In addition to abstract and concrete reference classes, Sather also supports **immutable** classes, which provide objects with value semantics. Many of the built-in types, such as INT and BOOL, are defined by immutable classes. Most of the restrictions imposed on the language to ensure safe sharing do not pertain to immutable objects, since they cannot be modified and are therefore free from interference problems.

For high-performance, and in the interest of providing type-safe container classes, Sather supports **parametrization**. A set which is parametrized over the type of elements, T, may be declared the following way:[3]

```
class SET{T} is
   insert(e:T) is ...
```

A unique feature of Sather is the support of iteration abstraction through **iterators** [22]. Iterators may be thought of as cursor objects that are provided directly through an object interface. Thus, a set class may provide an iterator to traverse its elements in its interface, rather than requiring a separate cursor object to perform the task. Iterators look similar to methods, but are distinguished by a terminating exclamation mark.

```
class SET{T} is
   elt!:T is ... return successive elements of the set
```

Iterators may be used in 'loop' constructs; the loop construct terminates execution whenever one of the iterators in it terminates:

```
contains(e:T):BOOL is
 Loop through all elements of the container until the iterator 'elt!' terminates
   loop
      element:T := elt!;
      if e = element then return true; end;
   end;
   return false;
end;
```

Implementation and Environment

Compilers, libraries, and graphical browsers for successive incarnations of Sather have been publicly distributed and maintained by ICSI[4] since 1991, and the compiler for Sather-K, a dialect of Sather [33], has also been distributed by the University of Karlsruhe, Germany. The ICSI Sather compiler generates portable C code. Sather has been used for teaching at the undergraduate and graduate levels in the US and Europe and for substantial research and implementation projects, including the compiler itself (which has hundreds of classes, over 170,000 lines of code). The compiler supports many common serial and parallel optimizations, and the resulting performance is comparable to that of C or C++ [8].

2 A RUNNING EXAMPLE: PARTICLE SIMULATION

As our running example, we consider the simulation of a system of particles. The particles are hard, frictionless spheres of fixed radius, and an initial position and velocity moving in a two dimensional box, colliding with the sides of the box and with each other. All collisions are assumed to be perfectly elastic. To simulate this system, the box is partitioned into a two-dimensional grid of rectangular areas. Each area contains a set of particles. Simulating an area for one time step involves:

- Updating the positions of all the particles within the area, handling any collisions in the process. This involves a pairwise i.e. O(n^2) computation to detect internal collisions.

---

3. In order to support separate compilation and type-checking of parametrized classes, type bounds are permitted on the type parameter.
4. The International Computer Science Institute, Berkeley, CA., US

- For any particles that might move into a neighboring area, we also check for collisions with particles in that neighboring area. Thus, there is an additional O(n), computation with all the balls in the neighbor into which they venture.
- Finally, at the end of an update cycle, all balls which have ventured outside their original area are transferred into their destination area

Collisions are simulated using the conservation of energy and momentum; since non-point particles are involved, the physics is somewhat complex and will not be described here. We ignore certain special cases, such as three way collisions; by making the time-step sufficiently small, such collisions may be made infrequent. We also assume that the granularity of the areas and the time-step are such that a particle can never cross a whole area in a single time step i.e. all exchanges of particles are between immediate neighbors only.

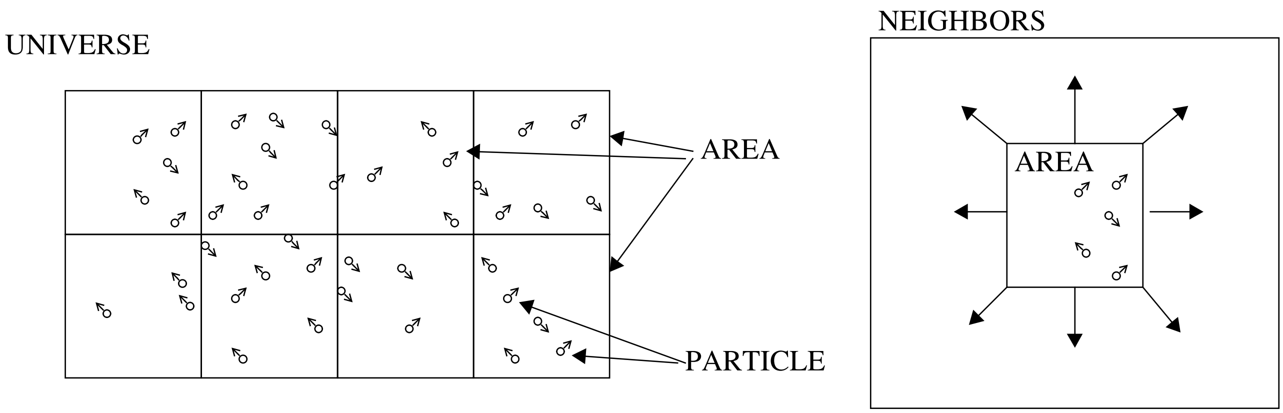

FIGURE 1. Particle Simulation

The main objects in the simulation are illustrated in Figure 1. The whole simulation is embodied by a UNIVERSE object, which is partitioned into AREA objects, each of which holds a set of PARTICLE objects. An AREA also keeps track of its neighborhood with a NEIGHBOUR object, which is also used to indicate whether the area borders on a wall.

3 Object Safety

The Goal

We are primarily concerned with a somewhat narrow meaning of object safety, namely, preserving the intended object semantics in the face of concurrency. By the object semantics we mean the set of allowed state transitions permitted by an object definition. Preserving object safety therefore means only permitting allowed transitions, even in a concurrent execution. In other words, ***in any parallel execution, all objects will only undergo the transitions allowed by a sequence of calls on the public interface of the class that defines the object.*** We refer to this property in the rest of this article as *object consistency* or *object safety.*

By object state transition we mean mutation of the object state. By the object state we mean all references within the object itself (i.e. its state variables) and, transitively, the state of all interior objects they refer to. The definition of interior objects is presented in section 5.2. Encapsulation of the object state requires controlled export of aliases to the deep state of the object that does not violate our object consistency goal. Alternatively, all object state transitions must be triggered by the invocations of the public interface methods rather than silent modifications of the objects transitively reachable through erroneously captured aliases. The main object consistency goal is achieved by a set of carefully designed method signature annotations and certain restrictions on the allowed call sequences to prevent the erroneous capture of interior state.

For instance, in our particle simulation example, we would like to avoid a situation when several areas erroneously try to simulate the same particles. Since particles in one area can collide with particles with other areas and move between areas, a common programming bug of capturing a reference to a particle not managed locally can result in a fairly expensive debugging effort. Our goal is to guarantee that this situation cannot happen at the programming language level in order to avoid a complex posterior runtime analysis.

At the very basic level, we would like to avoid concurrent mutations of the same memory locations with unpredictable results. We call such mutations basic data-races similar to [1]. While this is not sufficient to guarantee object consistency as defined in this section, this is the first step in that directions and the simplest object model that we will consider in Section 4 will do just that. However, we first review sequential consistency at the memory location level.

Sequential Consistency

While sequential consistency is central to our notion of safety, for performance reasons, all the SMP systems we are aware of do not provide sequential consistency at the hardware level. As do many others [1][2][9][10], we believe that presenting the programmer with a model that provides sequential consistency for the purposes of reasoning is indispensable for building large compositional concurrent systems.

The sequential consistency model was formally defined by Lamport:

> [A multiprocessor is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its programs. [17]

Sequential consistency maintains the memory behavior that is intuitively expected by most programmers. Each processor is required to issue memory operations in program order. Operations are serviced by memory one-at-a-time and appear to execute atomically with respect to other memory operations. The memory services operations from different processors based on an arbitrary, but fair global schedule. This leads to an arbitrary interleaving of operations from different processors into a single sequential order.

The benefits of such a model include at least the following: it matches most closely the intuitive assumptions that most programmers make about concurrent systems, eliminates basic data-races, and it makes programs portable across hardware platforms with different weak memory models [10].

Memory Consistency vs. Performance

There is a trade-off between performance and the strength of the memory consistency model. Achieving sequential consistency requires communication between processors about the state of shared data in order that all processors have a consistent view of the world. Reducing this communication by relaxing the consistency model enables hardware and software optimizations and can improve performance by 80-100% on modern platforms [10]. The challenge is to design a language that performs sufficient synchronization over shared data to provide the programmer with sequential consistency, while at the same time allowing the compiler and run-time to fully exploit the optimizations enabled by weaker consistency models between synchronization points.

It is also important that the object model *guarantee* such optimizations in the general case. In other words, the optimizations are not results of (often unpredictable) static compiler analysis, but are guaranteed by the underlying language design. We believe that the level of performance is important enough that we cannot rely on smart compilers to deliver this performance, as many other models have to (actors, behavior, strict active objects, etc.).

We believe that it is possible to design a language in which the annotations necessary for performance also reveal the program structure and modularity. In other words, we are interested in application-centric annotations that reveal the application structure, rather than system-centric annotations, which are concerned with hardware details. Application-centric annotations, which are related to the logical parallelism inherent in the application, can be important in designing the program and result in software that is easier to maintain and port. System-centric annotations, on the other hand, are related to the physical parallelism of the underlying system and distract from the program structure. System-centric annotations also result in non-portable code.

4 PROTECTING OBJECT STATE: THE CONSERVATIVE APPROACH

A trivial way to prevent conflicting data accesses is by requiring that all object accesses be serialized, i.e. only one method at a time is being executed on the object. A call arriving during the execution of a method is delayed by blocking the calling thread until the current method returns. Java's synchronized classes provide such mutual exclusion locking to protect object state.

4.1 Visitor/Mutator Annotations

A relaxation of this simple model is to permit multiple readers to access the object simultaneously. To avoid concurrent mutations of the same memory locations, it is sufficient that all calls on object methods that may mutate object state be mutually exclusive, while calls that merely examine or visit the object state may coexist with other visiting methods. We refer to this restriction as object-level visitor/mutator protection. Each object has a reader/writer lock associated with it, which is acquired by visitor or mutator methods.

In order to provide protection for the object state, all attribute accessor methods are implicitly annotated.

Basic Protection: Attribute Annotations

Provided that the primitive data access operations on the object (the reader and writer methods of the object attributes) are correctly annotated as visitors and mutators (respectively), sequentially consistent access is guaranteed. Since Sather defines implicit reader and writer methods associated with each attribute, we define all attribute reader methods to be visitors and all implicit writer methods to be mutators. This is sufficient to guarantee sequential consistency for this model. For instance, the attribute 'mass' in the PARTICLE class:

```
attr mass:FLT;
```

has implicit reader and writer methods:

```
mutate mass(f:FLT); -- Writer method
visit mass:FLT; -- Reader method
```

Since these accessor methods are used for all accesses to the attribute, the appropriate locking is guaranteed for all modifications.

Such basic attribute accessors annotations (that can be trivially generated by the compiler) are enough to avoid concurrent modifications of the same memory locations that is a source of many hard-to-find bugs especially well known to SMP programmers. Note however, that we are a long way from achieving object level consistency. For instance, the internal implementation of objects can be freely aliased and, as a result, objects can go through transitions that are not prescribed through the public interfaces of the outer layers. In other words, the goal of the deep state encapsulation is not addressed. Another problem with this basic model is performance - excessive locking at each attribute access essentially disables many hardware and compiler optimizations such as instruction reordering. It also results in repeated flushes of the store buffers and increased bus traffic of the bus based SMPs.

Coarsening Locking Granularity: Method Annotations

In this subsection we address the performance problem and make the first steps towards addressing the object consistency problem. We do so by allowing **visit** and **mutate** annotations of the class interface methods. Such methods are synchronized according to the concept of visitor/mutator protection. This has both semantic and performance benefits. In terms of performance, marking methods as visitor or mutators has the effect of coarsening the granularity of synchronization and reducing the number of times synchronization is required[5]. In terms of semantics, coarsening the granularity of the synchronization has the effect of preventing changes to the object for the duration of the method execution.

For example, consider a particle with mass and velocity, we could have a method that computes the kinetic energy of the particle:

```
attr mass:FLT;
```

5. Since the synchronization of object state accesses inside such methods may be trivially lifted by the compiler.

```
attr velocity:FLT;

visit kinetic_energy:FLT is return 0.5*mass * (velocity^2); end;
```

Even though the mass and velocity are individually protected against conflicting accesses, it is important to protect the entire 'kinetic_energy' method, so that the reader lock for the object is held for the duration of the method. If we did not do this, another thread could modify the velocity of the particle during the computation of the kinetic energy and the result returned by this method would be inconsistent. In Section 5 we note that particles are actually interior objects to an area, but our point about annotations nevertheless holds.

Example

As another example we consider the LOCATION class, which holds the new location (position and velocity) of a particle as shown in Figure 2. Consider the attribute 'xpos'. Sather provides implicit reader and writer methods

```
class LOCATION is
    private attr xpos,ypos:FLT;
    private attr xvelocity,yvelocity:FLT;

    create(x,y,xv,yv:FLT):LOCATION is ...

    mutate set_from(x,y,xv,yv:FLT) is
        xpos := x;
        ypos := y;
        xvelocity := xv;
        yvelocity := yv;
    end;

    x:FLT is return xpos end;
```

FIGURE 2. Visitor and Mutator annotations in class LOCATION

which are associated with setting and accessing the attribute value.

```
mutate xpos(val:FLT);
visit xpos:FLT;
```

This fine granularity of protection is insufficient in the case of the 'set_from' method - both position and velocity must be set simultaneously. Hence, we mark the method as a mutator, which ensures that the writer lock on the object is held for the duration of the method, thus preventing interleaved accesses to the object.

5 Shield and Interior Classes: *Encapsulation and Performance*

While locking at the level of individual object accesses is adequate to satisfy basic safety requirement at the memory location level (no concurrent modification of the same unprotected state), it suffers from drawbacks.

- Locking of frequently accessed methods can result in significant overhead and may be unnecessary if the object is always invoked within a safe context i.e. contained within some other object that provides the needed locking.
- Furthermore, though the locking guarantees sequential consistency in the basic memory location sense, it does not guarantee safety; it is up to the user to ensure that object-level safety is maintained. In the LOCATION example, for instance, if the 'set_from' method is not marked as a mutator, the object may well go through transitions that cannot arise by a sequence of calls on the object interface.

Many researchers have observed that in both serial and parallel object-based systems, groups of objects are often aggregated and for many semantic purposes it is convenient to think about the aggregate as a whole, rather than a composition of individual objects. For concurrent systems, such aggregation does not merely improve reasoning about programs, but can, in fact, affect the synchronization patterns.

Our solution is to relax the object model in ways similar to Emerald and Argus. The Sather model has two kinds of objects: shield objects and interior objects. The relationship between shield and interior objects is similar to that between guardians and their local state in Argus and global and local objects in Emerald. However, the important difference is that Sather *guarantees* by a combination of static and dynamic techniques that this relationship *always* holds (it is not just ensured by good programming style). This is important since, unlike Emerald and Argus, Sather allows dynamic sharing of objects between different object groups while maintaining proper protection.

The approach is somewhat similar to the idea of balloon types [3], but shield and interior objects allow further mechanisms like temporary sharing and transfer of interior object as described in Sections 6 and 7.

An executing Sather program consists of a collection of **shield** and **interior** objects that may send *synchronous* and *asynchronous* messages to each other. Objects are created at run-time and combine data representing the object state and program text.

5.1 Shield Objects

Shield objects provide protection for contained interior objects. A shield object together with interior objects that rely on it for protection compose an object domain or simply **domain**. Intra-object parallelism is permitted but constrained to prohibit conflicts between visiting and mutating methods.

Shield classes may block a caller of a method to provide protection in the face of shield access attempts. Intuitively, they may be viewed as “master” objects that protect a collection of interior objects. They are responsible for providing the protection for all interior objects within their domain. All interior objects are protected by an enclosing shield object.

Shield objects have the following properties:

- A shield class provides (blocking) protection against shield access.
- Intra-object concurrency is allowed, safe and is based on single writer/multiple reader paradigm.
- In the absence of pending synchronous calls from one domain into another domain, there are no inter-domain references to interior objects.
- To a first approximation, interior classes may not appear in a shield class public interface. This requirement will be relaxed in the next sections.

As an example, we consider an implementation of grid cell of our particle simulation example as a shield class protecting its particles:

```
shield class AREA is
    -- An internal representation of the collection of particles
    -- in the area, protected by this shield class
    private attr local_particles:SET{PARTICLE};

    -- print information about all particles currently in this area
    visit print is
        loop
            local_particles.elt!.print;
        end;
    end;
end
```

FIGURE 3. Shield class example

In this example, area is represented by a shield class AREA that hides the actual container ’local_particles’ from other areas. Method ’print’ simply iterates through all particles and prints their characteristics such as mass, velocity etc. We will consider the safety issues more closely in the following sections.

Visitor/Mutator Propagation in Interior Classes

Shield objects are responsible for providing protection for their state variables. Thus, as in the conservative model, the attribute accessor methods are implicitly annotated as visitors and mutators. Additionally, any method that invokes an attribute writer method of self is also a mutator, and any method that invokes an attribute reader method of self is also a visitor. The example in Figure 2 illustrates exactly the propagation of annotation. However, while this propagation of annotation is optional in the conservative model, it is enforced in our model as part of type-checking.

Thread Creation

The degree of concurrency can be increased by sending asynchronous messages to a shield object (but not interior objects) using a 'fork' expression. An executing asynchronous message is equivalent to a thread i.e. a potentially parallel computation with a separate locus of control but an identical notion of the address spaces and all system resources. The return value associated with a thread is stored in a future, from where it may be obtained later. For instance, in the following example, simulation threads one per area are created by the UNIVERSE class:.

```
shield class UNIVERSE is
    attr grid:ARRAY2{AREA}; -- the representation of a grid as a 2D array
    -- perform simulation in parallel by creating a simulation thread per area
    simulate is
        -- some initialization code
        loop
            -- iterate through all elements in the 2D array
            -- by calling elt! provided by ARRA2
            area:AREA:=grid.elt!;
            -- create a simulation thread
            f:FUTURE := fork area.simulate;
        end;
        -- some termination code
    end;
    -- other methods
end;
```

FIGURE 4. Thread Creation in Sather-2

All concurrency is generated by asynchronous method calls on shield objects, therefore, for completeness, the class containing the program's main routine must also be a shield class. Figure 5 presents a view of an executing Sather program.

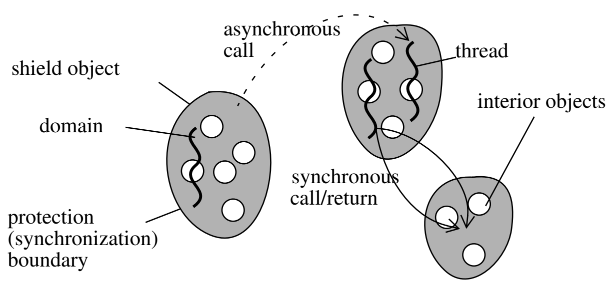

FIGURE 5. The Sather Object Model

5.2 Interior Objects

Interior classes rely on shield classes for protection in the face of concurrency. The language has been carefully designed to keep interior objects from silently escaping from one domain into another. This is essential since only shield objects control internal concurrency. In order to enforce the correct protection of interior class methods, all interior class methods must be correctly annotated.

Visitor/Mutator Propagation in Interior Classes

Since interior classes do not furnish protection on their own, they must propagate protection provided by the enclosing shield classes to their internal deep state. Protection for interior objects must always be provided at least by the first enclosing shield class and a set of methods annotations has been designed to ensure this statically for all interior classes during normal static type-checking phase. We now describe the rules that make such type-checking possible statically.

All methods that invoke a visitor method of an interior class must be marked as visitors. All methods that invoke a mutator method of an interior class must be marked as mutators. Thus, any shield or interior class method that calls a visitor method of an interior class must be marked as either a visitor or a mutator. Similarly, any shield or interior class method that calls a mutator method of a interior class must be marked as a mutator.

Example

In our example application, particles are interior objects which are held by an area, since particles usually cannot be accessed outside the area, and hence may be represented using an interior class.

```
interior class PARTICLE is

   readonly attr old_loc:OLD_LOCATION; -- location in the beginning of the time-step
   readonly attr new_loc:LOCATION; -- location at the end of the time-step
   readonly attr mass:FLT;
   readonly attr radius:FLT;

   visit collides_with(b2:PARTICLE):BOOL is
      Return true if a collision with 'b2' occurs
   end;

   mutate copy_new_to_old is
      old_loc.set_from(new_loc);
   end;

   visit collision_update(p2:PARTICLE) is
      Update the position of 'self' based on a collision with 'p2'
      new_x:FLT := ... old_loc.x* ...
      new_y:FLT := ... old_loc.y ...
      new_loc.set_from(new_x,new_y, new_vx,new_vy);
   end;
```

FIGURE 6. An Interior PARTICLE class

The ‘collides_with’ method only needs to read the old location of both particles and returns true if a collision will occur in the current time step. Thus, it is marked as a visitor, since it does not call any mutators on interior objects. At the end of a cycle, the new value is copied into the old location. This is a mutating operation since the old location is an interior class and a mutator operation, ‘set_from’, is called on it. The ‘collision_update’ method, on the other hand, only accesses the new location, which is a shielded object. Since the new location provides its own protection, no protection is required at the level of the particle itself. As may be seen, we have forced the level at which mutual exclusion is required down to the fine granularity of the location, rather than forcing the access of the particle (and thus the area) to be mutually exclusive.

5.3 Ensuring Object Consistency

In order to see that sequential consistency is ensured, we first note that

- All accesses of shield object state are safe (as before), since the reader and writer methods of shield classes are visitors and mutators, respectively, and the reader/writer synchronization succeeds before they are accessed.
- All accesses of interior object state must go through a shield class interface. All references to interior objects are only visible within their shield class and from other interior objects in the same domain. Thus, all method calls on a interior class must either occur from (a) the containing shield class or from (b) another interior class in the domain. Calls from another interior class in the domain must also occur from either (a) or (b). Since such a chain of calls (i.e. thread) may not originate in a interior class, by transitivity the chain of calls must originate in case (a).

Object level consistency is also ensured. The primitive attribute reader and writer access methods of interior classes are implicitly annotated. Our rule for visitor/mutator propagation ensures that at every call along a call path to an attribute reader or writer of a interior object either maintains or strengthens (when a call to a visitor occurs in a mutator method) the required protection. Thus, any method changing an object's state (its state variables and all reachable interior state) must be marked as a mutator and will be executed exclusively on that object. Hence, for each object the sequence of all state transitions during program execution can be described by a sequence of public method calls i.e. at any time a method is called the object is in a state that can be reached by a calling sequence on the public interface; no intermediate state is possible.

6 TEMPORARY SHARING OF INTERIOR OBJECTS

Unlike object models proposed by other researchers, inter-domain calls are also allowed. However, as with intra-object parallelism, a carefully crafted set of semantic constraints ensures safety by controlling the object reference aliasing. In this section, we consider temporary sharing of interior objects between object domains. In the following section we consider permanent transfers between object domains.

6.1 Exporting Interior Aliases

In order to relax the constraint on the containment of interior classes, we investigate the cases in which transferring interior classes between domains is safe. In general, there are two ways by which references to interior objects may be exported outside their domain. Both situations may arise from inter-domain calls:

- a public interface method of a shield class *takes interior objects as arguments*
- a public interface method of a shield class *returns a interior object*

Both situations are illustrated in the following figure.

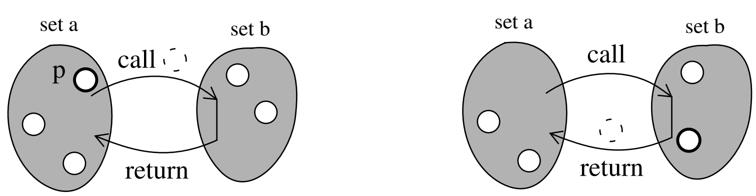

FIGURE 7. Interior objects and inter-domain calls

6.2 Object Capture

We first consider the problems that may arise when a interior object (or, more precisely, a reference to it) is passed as an argument in an inter-domain call. This case is graphically depicted in the left part of Figure 7. In the absence of any control over aliasing, area b can create a copy of the passed reference to object p and store it locally in area b, as an attribute of any object in area b domain. We call this *capture of a reference* or simply *capture*. Thus, even after call termination, area b will keep a reference to object p in area a as shown in Figure 8.

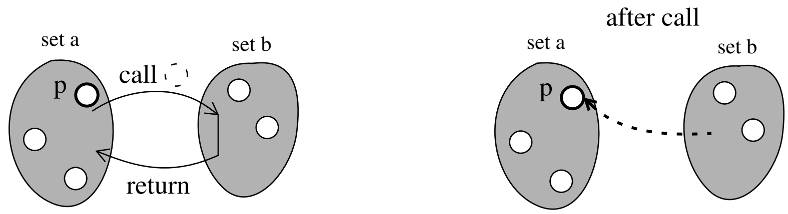

FIGURE 8. Reference capture and static inter-domain aliasing

The situation as depicted in the figure is inherently unsafe. After the call terminates, area b keeps a reference to an object in area a and can modify it concurrently with any other methods invoked on object p from area a.

While static analysis techniques can be used in order to detect problems with reference capture, we are dealing with a general aliasing problem. The problem is statically undecidable (although conservative heuristics do exist). This prompted other systems, such as Argus to disallow passing reference to interior data across domain boundaries. Instead, value semantics is assumed for all such calls and all objects are copied.

Such a solution may be fine for loosely coupled distributed system with high network latencies. However, for a high-performance system, especially if it is based on shared memory multiprocessors, the overhead of forcing value semantics for inter-domain calls restricts expressiveness of the language and has a negative performance impact.

6.3 Safe Exporting of Aliases

Instead of disallowing all inter-domain aliases, we disallow only the dangerous kinds that can compromise safety. Let us introduce some necessary terminology.

Aliasing Paths

Let *path to an object* or simply *path* be a sequence of variable names where each variable name binds an object and the last variable binds the object in question. The binding of each successive variable is evaluated in the context of the object bound by the previous variable. For interior objects, we are generally interested in paths that originate at shield objects. Inter-domain aliasing of an interior object exists if there are at least two paths to the object originating from different shield objects. Inter-domain aliasing is dynamic if two such paths exist only while the inter-domain call is in progress. Otherwise, aliasing is static.

Safe Paths

Not all kinds inter-domain aliasing are dangerous and therefore undesirable. For instance, consider a function in the AREA class that determines whether a function

```
visit collides_with_neighbor(p:PARTICLE):BOOL
```

must receive a reference to an element of another shield area as an argument. Disallowing this would unduly restrict language expressiveness.

Sather permits a subclass of inter-domain aliasing that can be easily statically proven to be dynamic (i.e. exists only while the method call is in progress). Such aliasing is safe - a caller maintains the necessary access permissions while the call is in progress and references are guaranteed not to be captured by the callee. This applies only to synchronous (non-threaded) calls. We will deal with asynchronous calls and semantic rules and restrictions necessary to maintain safety in the following sections.

The key observation is that for a reference to be captured, **a writer method for some attribute in an object in another domain must be executed**. Such writer methods (and possibly some other methods wrapped around it) are guaranteed to be marked as mutators. Hence, to eliminate all capturing of references to interior objects, it is sufficient (although not necessary) to disallow passing such references to mutator methods.

Rule for Temporary Sharing

In summary, these rules ensure the safety of inter-domain calls:

- Only visitor methods of shield class public interfaces may have interior (reference) objects as arguments.

Example

These rules significantly improve language expressiveness while not compromising safety. Considering our simulation, temporary transfer of a particle is required when checking for collisions with a neighboring area. During the handling of collisions, the only objects that may be updated are the new locations of the particles involved in the collisions. Since we designed LOCATION to be a shield type (which is natural since they may be modified from more than one area), the 'handle_neighbor_collisions' method need not propagate the mutation of the new location. Thus, the handle_neighbor_collisions method may be annotated as a visitor method, and an "outsider" particle may be safely passed into the method without any fear of it being captured.

```
visit handle_neighbor_collisions(outsider:PARTICLE) is
    loop b:PARTICLE := local_particles.elt!;
        if b.collides_with(outsider) then
            b.collision_update(outsider);
            outsider.collision_update(b);
        end;
    end;
end;
```

FIGURE 9. Temporary sharing: updating upon a collision

The performance improvement that results from permitting sharing of interior state, rather than requiring copying, depends entirely on the nature of the application. In the case of our particle simulation, copying is undesirable for design reasons - particles are naturally modeled as objects with a distinct identity which may be maintained through the lifetime of the application. Modeling them using value semantics requires us to maintain a separate notion of their object identity.

6.4 Ensuring Object Consistency

Object sharing means that two different domains may have aliases to the same interior state. While each domain will provide adequate protection on its own, accesses from the different domains may conflict with each other. Thus, both object consistency and even sequential consistency may be violated since encapsulation is violated. In order to maintain object consistency in the face of transient sharing of interior state it is necessary that

A] No unsafe operations occur during the transient sharing of state and

B] No interior state references remain after the sharing i.e. that the sharing of state is indeed transient.

As a preliminary, we reiterate the fact that all interior state is protected by a shield object, including any that comes in as an argument. Thus, a shield object method must be marked as a mutator if *any* interior objects are modified (including any that comes in as arguments). However, we only permit the passing of interior objects into visitor shield methods. Thus during such a visitor method, the callee domain may not modify any interior state, including the shared interior state. Furthermore, since the interior state belongs to the caller and is being accessed by the caller, the caller must also be marked as either a visitor or a mutator (it cannot be unmarked). In other words, the caller has, at the very least, obtained permission to visit the interior state. Since the call into the callee domain does not modify the shared state, any transitions in the shared state must take place in the caller. This reduces to the standard case for protecting interior state, as described in Section 5.3. Thus [A] is maintained. As described in Section 6.3, our rules prevent permanent capture of the shared state. Thus, [B] is also preserved and thus object consistency is preserved. Since object consistency implies sequential consistency, temporary sharing of state obviously preserves the sequential consistency of the programming model.

6.5 Why Not Use Copying?

The argument may be made that value semantics may be used whenever a domain wishes to access the interior state of another domain. There are two main arguments against this approach:

- Performance: value semantics may require the copying of a potentially large amount of interior state. If the sharing required is not computationally intensive, the cost of copying may dominate the performance. The extensive work done on compiler optimizations of such copying can certainly help; however, the performance may be fragile and unpredictable.
- More importantly, value semantics are fundamentally different than reference semantics, and the programmer may well desire one rather than the other in a particular context. While our model permits the programmer to use value semantics, it does not require him to do so, as is the case with Argus and Balloon types.

6.6 Why Not Use Shield Objects?

It is also possible to share internal state by protecting that state from conflicting accesses by making it a separate shield object. While this solution provides the right notion of object identity, it may unnecessarily affects performance. All accesses to the object, even within their own domain, will require locking. In our example, if particles are turned into concurrent objects, they will need to be locked on every access, including the safe accesses within their own area. This essentially forces us to use the conservative object model.

Thus, if possible, we wish to share interior objects when such sharing cannot compromise their safety. In cases where multiple domains may need to modify the same state, we can still resort to shielded objects, but this may be at a much finer granularity than would otherwise be needed.

In our example, for instance, the new location may be updated by the original area or by any of the neighbor areas and is therefore a shielded object. The particle, as a whole, however, including its old location which is used during the O(n^2) computations, need not be shielded. Thius, the old location is represented using an interior object, which permits us to greatly decrease the amount of locking required, since particles and old locations do not require locking.

7 TRANSFERRING INTERIOR OBJECTS

It is sometimes necessary to transfer objects between interior objects between different object domains. For the reasons mentioned in Section 6.5, copying is not desirable. The safety of an interior object that is transferred between domains may be compromised if more than one domains possesses an alias to the object - in this case, accesses to the object may pass through different interfaces, and therefore violate both object safety and even sequential consistency. Thus, in order to ensure object safety, it is necessary to ensure that at no time do both domains maintain pointers to the transferred interior state. This ensures that all interior accesses are controlled by a single shield interface, which provides adequate protection.

Sather provides a mechanism for transferring interior objects between different domains, such that the safety of the transfer may be dynamically determined. In order to permit the transfer of interior objects between domains, we first define *free* objects which may be *transferred* between domains. Since interior objects are always created within shield objects, at some point the connection between the interior group and the shield object must be severed. The severance of this connection is defined in two stages. We first define transferable groups of interior objects which are only reachable through a single external. We then define a free operation which destroys the last remaining external reference.

A interior object p defines a **transferrable** object group, consisting of itself and all object reachable from it, iff all paths to objects in the group go through p and there is only a single reference to p.

A transferable subgroup is **free** if there are no references to the root object p from any local variable or attribute of a shield class. Figure 10 displays the object groups associated with several free objects.

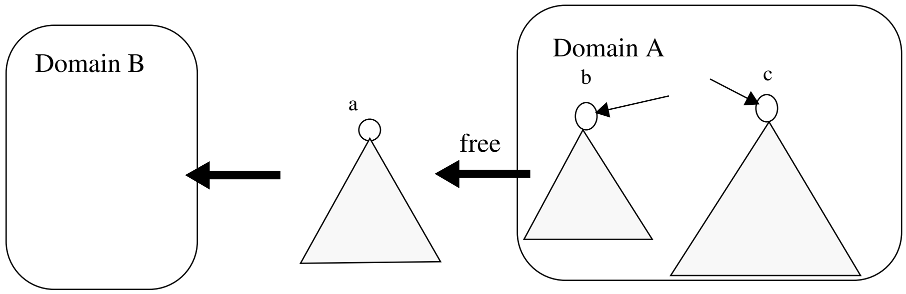

FIGURE 10. Free Object Groups

A transferable subgroup with root p is freed by the **free** operation applied to p, which releases the reference to the root object and verifies that the group reachable from the root object is indeed free. It returns the value of the reference, and may thus be used in calls as shown in the examples below. Objects *b* and *c* define transferable groups, while *a* is the root of a free group in Figure 10.

Free interior objects relax the last constraint on the interfaces of shield classes.

- Free objects may be passed as arguments to methods in shield classes.
- For the rules for visitor/mutator propagation applying free to a interior object is equivalent to calling a mutator on it.

7.1 The Transfer Mechanism

Transfer at the point of call is performed by using a mode specifier *free* with a method argument for both the method definition and the method call. The argument mode *free* applies the free operation by destroying the original reference to the free object. Any method that performs a transfer must be marked as a mutator method.

Transfer at the point of return is specified using the built-in *free* operation, which acts similarly to the free argument mode and sets a reference to a interior object passed as argument to void and returns a free object of the same type. It is an error if there exists a path from any shield class to any object in the object group to be released that does not contain a reference passed as argument to *free*. This last restriction ensures that the resulting object group is truly free and there are no references from original shield host left behind. The compiler emits code for a run-time check of this condition.

7.2 Run-time Checking

While all the safety mechanisms presented so far have been statically checkable, the transfer mechanism is not. Determining that an object group is transferable or free requires a run-time check of references, that may be performed with the aid of reference counts[6]. Reference counting will exact a certain performance penalty. The approach we pursue is to leave the decision up to the final user by providing a compiler option that will enable or disable the run-time check. This is similar to the Sather approach to other expensive tests such as array bounds and void dereference checking. In practice, the check is used during the debugging stage and not used in the production system. The following points may greatly reduce the performance of reference counting:

- Reference counting need not be performed at all in visitor methods. The interior state is not modified by a visitor method (i.e. no new references), and all local variables will disappear after the call terminates. If a vis-

6. The check basically ensures that the interior state being transferred is self-contained - all pointers to objects in the group come from other objects within the group. Details may be found in [12].

itor method calls a shield class mutator method, it cannot pass any interior state as an argument, since it cannot free the interior state (if interior state is freed then the method must be marked as a mutator).

- Reference counting need not be performed on local variables in certain mutator methods. Let us define a transferring method as a mutator method in which a free operation is performed or from which a transferring method is called (i.e. a transferring method is one from which a free operation is transitively reachable by a series of calls). Reference counting need only be performed on local variables of transferring methods. Reference counting must always be performed on attribute modifications, however, in all mutator methods.
- Private (object private) attributes that are never aliased may be transferred without any test.

Transferring may be viewed as an optimization over copying data between domains in the restricted case where the original domain no longer references the state being transferred.

7.3 Shield Class Interface Restrictions

In summary, then the restrictions on shield class methods are as follows:

- Visitor methods of a shield class may take any interior objects as arguments.
- Mutator methods of a shield class may only take free interior objects as arguments. Within the method body, only free objects may be obtained by calls to other shield class methods.

7.4 Example

The transfer mechanism is used within our particle simulation at the end of a time step to transfer particles that have moved out of their original area.

```
shield class AREA is
   mutate perform_transfers is
      ... identify a particle 't:PARTICLE' to transfer and the neighbor 'neighbor:AREA' to transfer it to
      particles.delete(t); -- Delete the particle from the local set of particles
      neighbor.receive_particle(free t);
      ....
   end;
```

FIGURE 11. Freeing a particle for transfer

As the code fragment in Figure 11 indicates, each area identifies the particles that need to be transferred. They are then deleted from their original area, and all internal references to them are destroyed. The particle is then 'free'd. The free particle may then be captured by a neighboring area, as shown in Figure 12and transferred to a neighbor area, where it may be captured.

```
   mutate receive_particle(free t:PARTICLE) is
      particles.insert(t);
   end;
```

FIGURE 12. Accepting a transferred particle

Transfer operations are generally useful in concurrent containers of interior objects. Such containers may be needed, for instance, in the work-bag paradigm to buffer small objects between activities in different but related domains.

8 SAFETY AND SUBTYPING

Our object model so far has been mainly concerned with ensuring safety in concrete classes. We now turn our attention to abstract classes (interfaces) and the need to ensure safe substitutability with subtyping. The subtyping rule in Sather is contravariant providing safe substitutability of classes by subclasses. This subtyping rule is extended to ensure better substitutability in the face of concurrency. Abstract classes may either be interior or shield. The type rules for abstractions are similar to the type rules for their concrete counter parts.

8.1 Abstract Interior Classes

Methods in abstract interior classes may be annotated as **visitable** or **mutable**. Methods marked visitable are considered visitor methods; protection for such methods must be propagated to all method calls, just as it is with concrete interior classes as described in Section 5.2. Abstract interior classes may only subtype from other abstract interior classes.

8.2 Abstract Shield Classes

Methods in abstract shield classes may also be unmarked or annotated as **visitable** or **mutable**. Methods marked visitable are considered visitor methods and methods marked mutable are considered mutator methods. Just as with concrete shield classes, method protection need not be propagated through calls. The same restrictions on public interfaces for concrete shield class apply to abstract shield classes.

Method arguments of abstract shield classes may also be marked as visitable or mutable. Return types may be unmarked or marked as **free.**

8.3 Subtyping between Shield and Interior Abstractions

Shield classes may subtype from abstract interior classes, provided, of course that they conform to the interior class interface. However, interior classes may not subtype from shield abstractions.

Informally, this subtyping constraint may be justified as follows. A visitor or a mutator method of a interior abstraction has the implicit precondition that the caller has become a visitor or mutator of the object before the method is called. The visitor/mutator propagation rule ensures the truth of this precondition. A visitor or mutator method of a shield class has the implicit precondition that the executing thread can become a visitor or mutator at some point in the future (i.e. doing so will not result in deadlock). Hence, the preconditions on interior class methods (i.e. that the caller is already a visitor or mutator) naturally imply the preconditions of shield classes (i.e. that it can become a visitor or mutator). The synchronization behavior of a method provides no specific postcondition guarantee.

Visitable vs. Visit

We choose to use the term visitable in abstractions to indicate that implementations may elect to not perform a synchronization, if they do not need to do so. Hence, the term visitable in the abstract class interface indicates that the user of the interface is not guaranteed that any synchronization will take place; the user must ensure that trying to become a visitor will not result in deadlock.

8.4 From Abstraction to Implementation

If the a method argument is annotated as a “visitable”, then an implementation subtype may annotate the argument as “visit”. From the point of view of preconditions, this means visitable implies visit. If an argument is of type visitable, the modifier states that it should be possible to become a visitor of the argument (or, alternately, that caller already is a visitor). The visit annotation has exactly the same requirement, and additionally, performs the action of actually taking the lock.

We have the following subtyping rules with respect to visit, mutate, visitable and mutable:

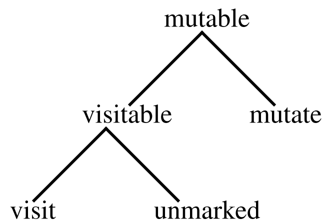

FIGURE 13. Extending Sather subtyping rules

8.5 Example

We use the above subtyping rules to illustrate how a more sophisticated version of the particle simulation algorithm can be built. Until now, the simulated universe has been subdivided into rectangular cells of equal size. However, for a non-uniform particle distributions more efficient adaptive meshes can be used. Thus, the universe is no longer subdivided into identical grid cells. To enable possible future algorithms refinements, we can declare an abstract class $AREA that captures area's essential interface:

```
abstract shield class $AREA is
    visitable print;        -- print information for local particles
    mutable simulate;   -- perform a simulation step
    mutable perform_transfers; -- transfer particles to the neighbors
    -- other interface signatures;
end;

-- A rectangular area - subtypes from the abstract interface $AREA
shield class RECT_AREA < $AREA is
    visit print is ... end;
    mutate perform_transfers is ... end;
    mutate simulate is ... end;
    -- other methods
end;
```

A class implementing the rectangular areas subtypes from the abstract class $AREA and the compiler uses the above extended subtyping rules during the type-checking phase.

9 PERFORMANCE

In this section, we consider the performance of the complete solution to the particle simulation problem in several different contexts. We concentrate specifically on the performance impact of the object models and the relevant trade-offs. All experiments were performed using a prototype Sather 2 compiler on the 8 cpu Sun Enterprise 5000 [30]. Each processor is a 167Mhz UltraSPARC-I with 16Kb of on-chip cache and 1Mb of external cache [28][29].

In all the experiments, the number of simulated particles was fixed at 250,000 and the size of the simulated area was also constant. However, we varied the granularity of the grid imposed on the area to observe the performance implications of different object models for threads of different granularities: from relatively coarse-grained to very fine-grained.

The speedups presented in Figure 14 are relative to the true sequential implementations of the base underlying algorithms - thread creation was replaced with direct function calls and all synchronization (including protection locking) was removed. The base sequential implementations use the same grids as the corresponding parallel implementations.

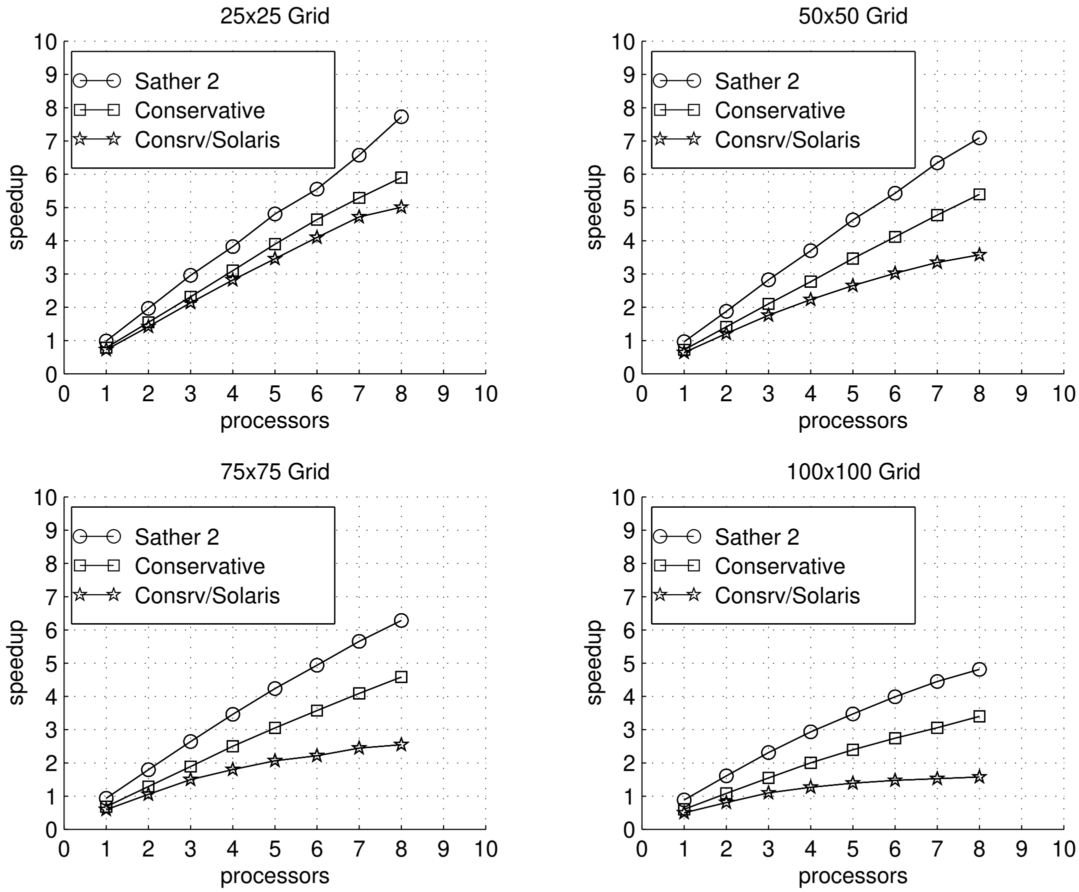

FIGURE 14. Particle Simulation Performance on Sun E5000

Since in our algorithm a separate thread is created to simulate each grid cell, the grid granularity determines the total number of threads used in the simulation. For instance, 10,000 threads were created in parallel runs for a 100x100 grid.

The number of simulated particles is fixed at 250,000. Therefore, for a uniform initial distribution, the number of particles simulated by a single thread ranges from 400 for the 25x25 grid to only 25 for the 100x100 grid. This yields several different thread granularity contexts - from relatively coarse- to relatively fine-grained.

For each grid size, Figure 14 presents speedup curves for three different implementation of the particle simulation:

- The Sather 2 implementation based on the object model and the solution outline presented in this paper.
- An implementation that uses conservative locking for all objects (i.e. all classes are shield classes).
- Conservative as above, but uses the vendor-provided synchronization primitives instead of the Sather runtime.

We compare the performance of the Sather 2 implementation with two conservative implementations. Both implementation share the same source code but were compiled using different compilation options. In the first case, the compiler utilized Active Threads, a portable fine-grained thread system designed as a compilation target for Sather [34][35]. The second case is based on Solaris Threads, a native vendor-provided thread package for Sun Enterprise 5000 [31].

The Sather 2 solutions display fairly good performance and scalability even for relatively large numbers of fairly fine-grained threads. For instance, in the 75x75 case, there are 5625 threads and each thread initially simulates only

44 particles. The observed speedups for this case are between 6 and 7 on the 8 cpu E5000. For coarser granularities, Sather 2 demonstrates close to perfect speedups.

The first conservative implementation (based on Active Threads locks) is roughly 40% slower than the Sather 2 implementation. Excessive locking increases the total synchronization overhead and disables many hardware and compiler optimizations that exploit the weak hardware memory consistency model supported by the E5000. Conservative solutions are also very sensitive towards the system-level details of the reader/writer lock implementation. For instance, acquiring an uncontested lock for reading using Active Threads takes 0.2µs. An equivalent operation for Solaris Threads takes 1.0µs. This difference is largely responsible for smaller speedups of the Solaris based implementation of the conservative model.

The speedups demonstrated for the finest thread granularities are somewhat smaller for the Sather model and significantly smaller for the conservative models. For instance, the Solaris based implementation for the 100x100 grid (25 particles per thread initially) displays a speedup of only 1.6 on 8 processors. The increased total synchronization overhead is only partially responsible for this performance degradation. The other factor that gains importance for fine granularity threads is locality at all levels of the memory hierarchy. During each simulation step, threads must load the necessary data into the cache. Cache reload overhead is amortized over the total number of instructions executed by a thread during the course of the time step simulation. If threads simulate only several dozens of balls, they simply do not execute long enough for the amortization effects to be significant. Locality issues in the context of Active Threads and different locality enhancing thread scheduling policies are discussed further in [34].

For completeness, we also compare the performance of Sather with that of the vendor-provided thread library on a set of microbenchmarks (Figure 15)

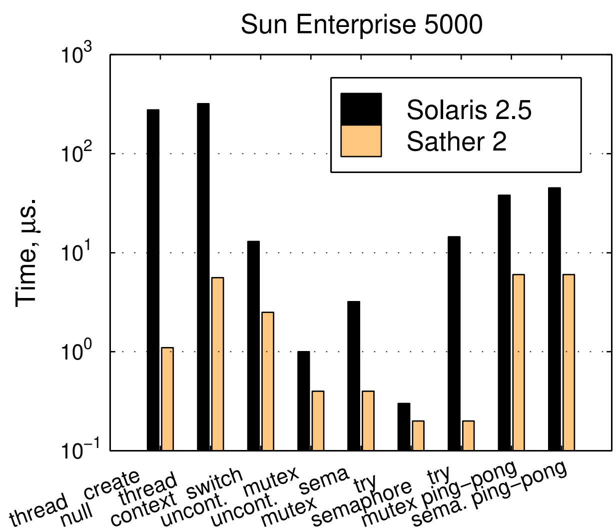

FIGURE 15. Sather 2 vs. vendor thread library.

The benchmark operations are the following:

- The *thread create* operation, including allocation of a thread stack but excluding context switch time and execution of a newly created thread.
- The *null thread* benchmark measures the entire runtime of thread that performs a null call from creation to termination.
- *Context switch* time is measured by having a thread yield execution to another thread.
- *Uncontested mutex* and *uncontested semaphore* benchmarks time mutex lock and semaphore wait operations in the absence of contention.
- Mutex and semaphore *try* benchmarks measure the overhead of the corresponding non-blocking operations.

- Mutex and semaphore *ping-pong* operations repeatedly synchronize two threads with one another in a manner similar to that used to measure synchronization cost of Solaris Threads in [23]. Ping-pong timings include the synchronization of two threads. Per thread synchronization overhead as reported in [23] is exactly one half of the reported numbers.

Having an efficient Sather runtime implementation has proved to be an important factor in evaluating the performance implications of different object models.

10 CONCLUSION

The problem of deep object state encapsulation has not received enough attention in the previous research to enable practical solutions in existing object-oriented environments. Previous design proposals for sequential languages either concentrated on limited forms of object sharing or were targeted at disallowing static sharing while permitting unrestricted dynamic sharing. In concurrent environments, safe sharing of objects gains even greater importance because of the prohibitive debugging overheads and qualitatively new kinds of non-deterministic object access conflicts. Both static and dynamic aliasing can be inherently unsafe and must be addresses by the underlying object model.

We have presented a concurrent object-oriented language mechanism that ensures safety in the presence of object sharing while maintaining efficiency. Safety and efficiency are both vital to the class of explicitly parallel object-oriented languages and various inherent trade-offs must be considered and carefully evaluated for any viable object model design. The safe object sharing mechanism described in this paper lead to the creation of Sather-2, a new high-performance object-oriented language that ensures object safety in concurrent environments. The language was designed to specifically target modern computing platforms such as commodity SMPs (with different weak hardware consistency models), and clusters of SMPs. A prototype compiler for the language has been used to evaluate performance impacts of different object model designs on several applications and microbenchmarks. Performance measurements substantiated our claim that the goals of safety and efficiency can be simultaneously reached in a single parallel object-oriented language.

Future work is concerned with developing object-oriented methodologies for building safe, high-performance parallel libraries that will exploit further the benefits of the Sather-2 object model. The compiler is undergoing extensive tests before a public release.

11 REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, R. H. B. Netzer, Detecting Data Races on Weak Memory Systems, 18th Annual International Symposium on Computer Architecture, June 1991.

[2] S. V. Adve, K. Gharachorloo, Shared Memory Consistency Models: A Tutorial. IEEE Computer, December 1996, pp. 66-76.

[3] P.S. Almeida, Balloon Types: Controlling Sharing of State in Data Types, Proceedings of ECOOP ’97, pp. 32-59, 1997.

[4] G. Agha, Actors, A Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[5] American National Standards Institute, Inc., The Programming Language Ada Reference Manual, LNCS 155, Springer-Verlag, 1983.

[6] D. Berg. Java Threads, A Whitepaper. Sun Microsystems, March 1996.

[7] A. Black, N. Hutchinson, E. Jul, H. Levy, Object Structures in the Emerald System. OOPSLA ‘86, ACM SIGPLAN Notices, vol 21, no 11, pp. 78-86, Nov 1986.

[8] C. Fleiner, Parallel Optimizations. Advanced Constructs and Compiler Optimizations for a Parallel , Object-Oriented Shared Memory Language running on a Distributed System. Ph. D. Thesis, Institute of Informatics of the University of Fribourg, Switzerland, 1997.

[9] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, M. D. Hill, Programming for Different Memory Consistency Models. Journal of Parallel and Distributed Computing 15, 1992, 399-407.

[10] K. Gharachorloo, Memory Consistency Models for Shared-Memory Multiprocessors. Ph.D. Thesis. Department of Electrical Engineering, Stanford University, 1996.

[11] B. Gomes, D. P. Stoutamire, B. Weissman, H. Klawitter, Sather 1.1 Language Essentials. International Computer Science Institute. Available at http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/contents.html.

[12] B. Gomes. J. Quittek, W. Lowe, B. Weissman, Sather-2: A Language Design for Safe, High-Performance Computing. International Computer Science Institute TR-97-035 1997.

[13] J. Hogg, Islands: Aliasing Protection in Object-Oriented Languages, OOPSLA 1991, pp. 271-285.

[14] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. Follow-up report on ECOOP'91 workshop W3: Object-oriented formal methods. OOPS Messanger, 3(2):11-16, April 1992.

[15] Intel Corporation, Pentium Pro Family Developer's Manual. Volume 3: Operating System Writer's Guide. Order Number 242692. December 1995.

[16] W. Kim. Thal: An Actor System for Efficient and Scalable Concurrent Computing. Ph. D. Thesis. University of Illinois at Urbana-Champaign, 1997.

[17] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. IEEE Transactions on Computers, C-28(9):690-691, September 1979.

[18] D. Lea, Concurrent Programming in Java, Addison-Wesley, Reading, Massachusetts, 1997.

[19] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert, Abstraction Mechanisms in CLU, CACM, August 1977.

[20] B. Liskov, R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM TOPLAS, Vol. 5, No. 3, July 1983, pp. 381-404.

[21] C. May, E. Silha, R. Simpson, H, Warren, Eds., The PowerPC Architecture: A Specification for a New Family of RISC Processors. Morgan Kaufman Publishers Inc., 1994.

[22] S. Murer, S. Omohundro, D. Stoutamire and C. Szyperski , Iteration Abstraction in Sather. Transactions on Programming Languages and Systems, Vol. 18, No. 1, Jan 1996 p. 1-15.

[23] M. L. Powell, S. R. Kleinman, S. Barton, D. Shah, D. Stein, M. Weeks. SunOS 5.0 Multithreaded Architecture. A White Paper. Sun Microsystems, 1991.

[24] J.W. Quittek, B. Weissman, Efficient Extensible Synchronization in Sather, will appear in: Proceedings of the 1997 International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE '97), Springer-Verlag, 1997.

[25] The Institute of Electrical and Electronics Engineers. Portable Operating System Interface (POSIX) - Part 1: Amendment 2: Threads Extensions [C Language]. POSIX P1003.4a/D7. April, 1993

[26] R. Sites, Ed., Alpha Architecture Reference Manual. Digital Press 1992.

[27] The SPARC Architecture Manual. Version 8. SPARC International, Inc., Prentice Hall, 1992

[28] SPARC International Inc., The SPARC Architecture Manual. Version 9. Eds. D. L. Weaver, T. Germond. Prentic Hall, 1994

[29] Sun Microelectronics. UltraSPARC User's Manual, 1996.

[30] Sun Microsystems, The Ultra Enterprise 1 and 2 Server Architecture. Technical White Paper. April 1996.

[31] D. Stein, D. Shah, Implementing Lightweight Threads. Summer '92 USENIX, San Antonio, Tx

[32] D.P. Stoutamire, S. Omohundro, Sather 1.1 Specification. International Computer Science Institute, Berkeley Ca. Technical Report TR-96-012.

[33] D. Stoutamire, W. Zimmermann, and M. Trapp, An Analysis of the Divergence of Two Sather Dialects. International Computer Science Institute TR-96-037, 1996.

[34] B. Weissman, Active Threads: an Extensible and Portable Light-Weight Thread System. International Computer Science Institute TR-97-036 1997.

[35] B. Weissman, B. Gomes, J. W. Quittek, M. Holtkamp, Efficient Fine-Grain Thread Migration with Active Threads. Submitted to the 12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)